# SyFER-MLIR: Integrating Fully Homomorphic Encryption Into the MLIR Compiler Framework

Sanath Govindarajan
*MIT PRIMES*
*University of Texas at Austin*
sgovindarajan@utexas.edu

William S. Moses
*MIT CSAIL*
wmoses@mit.edu

***Abstract*—Fully homomorphic encryption opens up the possibility of secure computation on private data. However, fully homomorphic encryption is limited by its speed and the fact that arbitrary computations must be represented by combinations of primitive operations, such as addition, multiplication, and binary gates. Integrating FHE into the MLIR compiler infrastructure allows it to be automatically optimized at many different levels and will allow any program which compiles into MLIR to be modified to be encrypted by simply passing another flag into the compiler. The process of compiling into an intermediate representation and dynamically generating the encrypted program, rather than calling functions from a library, also allows for optimizations across multiple operations, such as rewriting a DAG of operations to run faster and removing unnecessary operations.**

## I. Introduction

As the world has become increasingly digitized, concerns for privacy have also risen. Computation on sensitive data such as medical records, messages, and locations is being deferred to companies with large amounts of computing power and proprietary algorithms, and therefore, the need for security parallels the ever-increasing need for computation in fields like genomics, health care, and national security [2].

For example, consider a hospital which would like to submit the records of its patients to a third-party organization to analyze its patients for common risk factors for a certain disease. Due to legal restrictions about the release of health records, as well as ethical concerns regarding the privacy of its patients, such an analysis may be difficult or even impossible to request.

### A. Homomorphic Encryption

Homomorphic encryption is a type of asymmetric encryption in which secure computations can be performed on unknown data. In other words, an algorithm can be run on encrypted data, and can produce an encrypted result. Consider the following example: Suppose that a sender has a private message $\mu$ and they wish to know the result of a function $\mathrm{f}(x)$ which only the receiver can compute. The sender may or may not know f. First, the sender creates a secret key sk and a public key pk. They then use an encryption function $\mathrm{Encrypt}(\mathrm{pk}, \mu) \to \psi$ to encrypt their message $\mu$. The receiver receives $\psi$ and performs an encrypted computation $\mathrm{F}(\psi)$, where F is an *encrypted analogue* of f. They send this back to the sender, who decrypts $\mathrm{F}(\psi)$ to $\mathrm{f}(\mu)$ (a homomorphic encryption scheme is one where, by definition, $\mathrm{F}(\psi) = \mathrm{f}(\mu)$). So, although the receiver did the computation, neither the message nor the result was revealed to them.

There are four kinds of homomorphic encryption: partially homomorphic, somewhat homomorphic, leveled fully homomorphic, and fully homomorphic encryption [1]. In partially homomorphic encryption, the set of possible f is limited by the set of possible primitives: some schemes only allow addition, while others only allow multiplication. In somewhat homomorphic encryption (SWHE), the set of possible f is limited by the number of operations: only small circuits can be evaluated. Leveled fully homomorphic encryption is similar to SWHE, except that circuits with a bounded depth can be evaluated, instead of just small circuits. And in fully homomorphic encryption, or FHE, any arbitrary function f has an encrypted analogue F [18]. Somewhat homomorphic encryption, leveled fully homomorphic encryption, and homomorphic encryption, can all be proven complete as long as either addition and multiplication are both homomorphic, or there exists an encrypted NAND gate, while they differ in their guarantees on the increase of error after several computations [1]. It should be noted that a leveled fully homomorphic scheme can be transformed into a FHE scheme using a process known as *bootstrapping* [8].

(Leveled) fully homomorphic encryption aims to solve the problem of arbitrary computation by an un-trusted party on private data. The sender knows the original message and the result, while the receiver (who performs the computation) knows neither.

### B. Related Work: Existing FHE Libraries

Currently, there exist many (leveled) fully homomorphic encryption schemes which rely on the hardness of the learning-with-errors problem (LWE) [17] or its ring variant RLWE [14]. Popular schemes include the Brakerski-Gentry-Vaikuntanathan (BGV) [3], Fan-Vercauten (B/FV) [7], the Residue Number System (RNS) variant of B/FV [10], and Cheon-Kim-Kim-Song (CKKS) [5] schemes. A less popular scheme is the Gentry-Sahai-Waters (GSW) scheme, which requires that its message is encoded in binary [8].

There also exist many implementations of these schemes, including Microsoft SEAL [4], PALISADE [15], HElib [11], and TFHE [6]. These libraries all expose a similar set of

Example 1: Common Subexpression Elimination (CSE) - a high-level optimization

```
int func(int a, int b, int c, int d) {
    int e = a * b + c;
    int f = a * b * d;
    return (e + f);
}

int fast_func(int a, int b, int c, int d) {
    int ab = a * b;
    int e = ab + c;
    int f = ab * d;
    return e + f;
}
```

Example 2: Loop Tiling - a low-level optimization

```
int sum_loops(int *arr) {
    int sum = 0;
    for (int i = 0; i < 128; i++) {
        for (int j = 0; j < 128; j++) {
            sum += arr[i][j];
        }
    }
    return sum
}

int fast_sum_loops(int *arr) {
    int sum = 0;
    for (int i = 0; i < 128; i += 16) {
        for (int j = 0; j < 128; j += 16) {
            for (int k = i; k < 16; k++) {
                for (int l = j; l < 16; l++) {
                    sum += arr[k][l];
                }
            }
        }
    }
    return sum
}
```

Fig. 1. A few of the optimizations that can be performed by MLIR, represented using C code. In reality, these optimizations happen on code in the MLIR intermediate representation. See Figure 4 for an actual example written in MLIR.

functions to the user: typically one function per primitive (such as addition or multiplication) [18]. One library which takes a different approach is CrypTen [9], which exposes encrypted tensor operations using secure multiparty computation.

The main tradeoff in homomorphic encryption (especially FHE) is decreased speed and increased memory usage in exchange for increased security. A single encrypted bit takes several kilobytes to store, and a single homomorphic AND gate takes multiple milliseconds to compute, compared to a single clock cycle for a native AND gate. Conventional wisdom states that fully homomorphic encryption will always be dramatically slower as a result of the overhead necessary to ensure secure computation. Diving in deeper, however, a significant part of this overhead can be explained by the inability of current tools to optimize homomorphic programs. For example, many common optimizations like common subexpression elimination and loop tiling (see Figure 1) cannot be performed by a regular FHE library on encrypted programs, but they can be performed by MLIR [13] and other compilers on regular programs. In fact, there are four main limitations of current FHE libraries:

1) They can only perform a limited set of cross-operation optimizations. All current FHE libraries provide a set of homomorphic primitives, such as Add $(x, y, params)$ and Multiply $(x, y, params)$ (or their respective operator overloads) [4] [15] [11], or in the case of GSW and its variants, NOT $(x, params)$, AND $(x, y, params)$, and the rest of the binary gates [6]. And each operation is separately optimized. But these libraries cannot look across multiple operations to remove or simplify intermediate instructions, for the most part.
2) They cannot perform rewrites and other high-level optimizations, in which, for example, NAND $(x, x, params)$ is replaced with the much faster NOT $(x, params)$ or Multiply $(x, 1, params)$ is replaced with $x$.
3) They do not perform low-level optimizations such as loop nest optimization and loop scheduling.
4) They are not modular. If a user wants to use a different encryption scheme, they must rewrite their code, or even switch to a different library. Additionally, there is no modularity in programming languages (most of them must be accessed in C++, although SEAL can be used in C++, C#/F#, Python, and JavaScript). And finally, there is no modularity in optimizations. The user cannot choose which optimizations to apply and which ones to avoid based on their own circumstances.

### C. Our Contribution

In order to overcome all of these limitations, we present SyFER-MLIR, a Synthesizer of Fast Encrypted Routines, which uses the MLIR compiler framework to generate encrypted programs. The previous version of SyFER interfaced with Halide [16] to perform cross-operation optimizations and rewrites on the GSW scheme, and the current version uses MLIR to vastly increase low-level optimizations, modularity at every level, and ease of use.

With SyFER-MLIR, instead of having to manually translate a plaintext function f into an encrypted function F with the help of a library, the function f is kept as-is, written in (a subset of) any language that compiles to MLIR, such as C, C++, Fortran, or TensorFlow. When it is compiled by MLIR, it is automatically translated into an intermediate representation known as the Standard Dialect. A subset of the Standard Dialect is then automatically translated into any one of a number of FHE dialects (currently, GSW and B/FV are implemented). Cross-operation optimizations and rewrites are applied, after which point the result is passed back into MLIR, optimized further at the low-level, and then passed into LLVM for the lowest-level optimizations [12].

Unlike current FHE libraries, SyFER-MLIR reads in an *entire*, arbitrary program, and encrypts it all at once. The program can be compiled to any one of a number of targets, such as serial code, CPU parallel code, and GPU parallel code in any architecture supported by LLVM. The specific optimizations performed and omitted by SyFER-MLIR can be tweaked by the user in the command line, so that different sets

of optimizations can be performed for different programs and specific machines and use cases, all without actually modifying a single line of code. The specific parameters of the FHE scheme with which they will need their program to work can also be directly modified in the command line, without any modifications to the program itself.

## II. Optimizations

### A. MLIR and Progressive Lowering

MLIR, and consequently, SyFER-MLIR, can perform cross-operation optimizations, high-level optimizations, and low-level optimizations using a process called *progressive lowering* [13]. In a traditional compiler, a programming language is translated directly into low-level instructions. However, modern compilers use several intermediate representations to perform optimizations at different levels of abstraction. At each lowering step, a higher-level IR is optimized and then converted into a slightly lower-level IR. MLIR allows this process to be easily modified and extended with its concept of *dialects*. The dialects all have very similar syntax, which allows different dialects to be processed with the same tool (MLIR) and easily converted from one into another. However, the dialects have very different operations, with some dialects dealing with high-level matrix and tensor math, while other dialects deal with low-level loop optimizations. The operations themselves are completely opaque to MLIR, and are converted from one dialect to another using lowering rules which are defined separately. Dialects can also be mixed within the same MLIR file, which allows, for example, the GSW dialect to lower to a mix of LinAlg, Affine, SCF, and Standard dialects in just two steps. The only requirement is that the complexity of the code should be monotonically decreasing between successive lowering steps.

By creating a dialect for a FHE scheme instead of writing a library, we can take advantage of progressive lowering to automatically generate low-level instructions like addition and multiplication from a complicated program. Then, these instructions are automatically converted into their encrypted analogues using custom lowering rules. Finally, the encrypted analogues are lowered back into executable instructions like normal.

Several lowering passes are needed to fully lower a FHE dialect into LLVM IR. Figure 2 is a simplified diagram showing all the dialects which a snippet of code travels through when being lowered through SyFER-MLIR into LLVM-IR code. The dialects are colored according to their complexity, with more complex dialects appearing on the red side of the spectrum while less complex dialects are on the blue side. Figure 3 shows the complete set of passes needed to lower the GSW and B/FV dialects without any optimizations. If further optimizations are needed, an optimization pass can be added between any two lowering passes in the pipeline. Note that the first lowering pass in both the GSW and B/FV pipelines converts each dialect into a "simple" version of itself, lowering complicated GSW operations into simple GSW operations, and similarly for B/FV. The simple dialects are then lowered into other dialects.

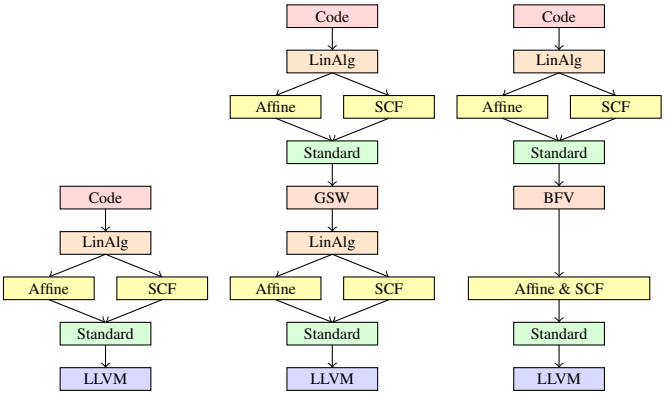


Fig. 2. Simplified diagram of progressive lowering of code through the regular, GSW, and B/FV pipelines. Note that different dialects are often present within the same file, which is not illustrated. GPU scheduling and further loop optimizations are possible but not illustrated.

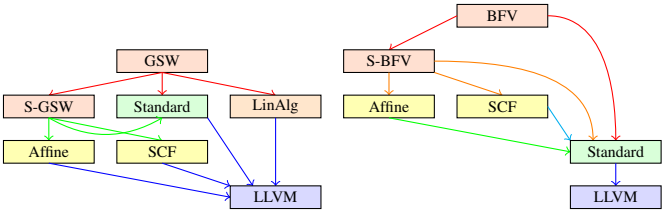


Fig. 3. Diagram of progressive lowering from the GSW and B/FV dialects into LLVM-IR. On the GSW side, there are three passes: GSW to Simple GSW (GSW dialect → mix of Simple GSW, LinAlg, Standard), Simple GSW to Loops (Simple GSW → Affine, SCF, Standard), and Linalg to LLVM (Linalg, Affine, SCF, and Standard → LLVM). On the B/FV side, there are five passes: BFV to Simple BFV (BFV → Simple BFV, Standard), Simple BFV to Loops (Simple BFV → Affine, SCF, Standard), Lower Affine (Affine → Standard), Loops to Standard (SCF → Standard), and Standard to LLVM (Standard → LLVM). Each set of arrows of the same color represents a pass, with earlier passes appearing on the red side of the spectrum and later passes appearing on the blue side.

### B. Circuit Graph Rewrites

Every encrypted function in a FHE scheme can be represented as a directed acyclic graph (DAG) of primitives in which each node is a primitive and each edge represents the output of one primitive being passed in as the input of another. In the case of the GSW scheme, these primitives are binary gates, and in the case of other schemes like B/FV, these primitives are addition and multiplication.

Rewrites allow us to replace more time-consuming operations with semantically equivalent faster operations. As previously mentioned, in the GSW scheme, a homomorphic NOT gate is much faster than a homomorphic NAND gate, so we can define a rewrite rule to replace NAND $(x, x, params)$ with NOT $(x, params)$. We can also aggressively eliminate dead code within individual circuits like encrypted adders.

Such optimizations would be impossible with a standard FHE library.

In addition to these simple rewrites, there are also more complicated rewrites generated by SyFER using techniques like trimming, removing constants, collapsing NOT chains, and optimizing 2-input and 3-input subgraphs. These optimizations can also be performed in SyFER-MLIR using the DAG Rewriter Infrastructure.

### C. Cross-Operation Optimizations

Cross-operation optimizations allow us to remove unnecessary or canceling lower-level instructions which span multiple operations. For example, in the B/FV scheme (whose ciphertexts are members of an anticyclic polynomial ring), two INTT's added together can be replaced by a single INTT after the addition. Example pseudocode ($a$, $b$, $d$, and $e$ are polynomials initially in coefficient form, $+$ represents polynomial addition, and $\cdot$ represents element-wise multiplication).

**Algorithm 1** sum of two products of polynomials (unoptimized)

1: $a \leftarrow NTT(a)$
2: $b \leftarrow NTT(b)$
3: $c \leftarrow a \cdot b$
4: $c \leftarrow INTT(c)$
5: $d \leftarrow NTT(d)$
6: $e \leftarrow NTT(e)$
7: $f \leftarrow d \cdot e$
8: $f \leftarrow INTT(f)$
9: $g \leftarrow c + f$
10: **return** $g$

can be optimized to

**Algorithm 2** sum of two products of polynomials (optimized)

1: $a \leftarrow NTT(a)$
2: $b \leftarrow NTT(b)$
3: $c \leftarrow a \cdot b$
4: $d \leftarrow NTT(d)$
5: $e \leftarrow NTT(e)$
6: $f \leftarrow d \cdot e$
7: $g \leftarrow c + f$
8: $g \leftarrow INTT(g)$
9: **return** $g$

In practice, the INTT's, element-wise multiplications, and polynomial additions are part of the "simple BFV" dialect, and would have been lowered from multiple "complicated BFV operations". Such optimizations could occur across multiple original operations. Libraries like Microsoft SEAL and PALISADE can perform this particular optimization by keeping track of the state of each polynomial (whether it is in coefficient form or NTT form), but MLIR provides a general solution for all possible cross-operation optimizations.

Memory deallocation of every matrix (in the GSW scheme) and polynomial (in the B/FV scheme) can also be optimized to occur at precisely optimal times across multiple original operations using MLIR's buffer placement feature.

Unoptimized:

```
#map0 = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<() -> (0)>
#map2 = affine_map<()[s0] -> (s0)>


module {
  func @main() {
    %c0_i64 = constant 0 : i64
    %c128 = constant 128 : index
    %0 = alloc(%c128, %c128) : memref<?x?xi64>
    affine.for %arg0 = 0 to %c128 {
      affine.for %arg1 = 0 to %c128 {
        affine.store %c0_i64, %0[%arg0, %arg1] : memref<?x?xi64>
      }
    }
    return
  }
}
```

Optimized:

```
#map0 = affine_map<(d0, d1) -> (d0, d1)>
#map1 = affine_map<(d0) -> (d0)>
#map2 = affine_map<(d0) -> (d0 + 16)>
#map3 = affine_map<() -> (0)>
#map4 = affine_map<()[s0] -> (s0)>


module {
  func @main() {
    %c0_i64 = constant 0 : i64
    %c128 = constant 128 : index
    %0 = alloc(%c128, %c128) : memref<?x?xi64>
    affine.for %arg0 = 0 to %c128 step 16 {
      affine.for %arg1 = 0 to %c128 step 16 {
        affine.for %arg2 = #map1(%arg0) to #map2(%arg0) {
          affine.for %arg3 = #map1(%arg1) to #map2(%arg1) {
            affine.store %c0_i64, %0[%arg2, %arg3] : memref<?x?xi64>
          }
        }
      }
    }
    return
  }
}
```

Fig. 4. Example of MLIR's affine loop tiling optimization

### D. Low-Level Optimizations

After lowering through an FHE dialect, the resulting code is further optimized by MLIR by passing through various dialects like linalg (linear algebra) and affine (affine loops and expressions). These dialects can further optimize the the loops by doing low-level scheduling optimizations like loop tiling and unrolling (see Figure 4). There are also optional passes to parallelize and/or GPU-schedule the loops. Along with the benefit of modularity of optimizations, this approach also decouples algorithm definition from execution strategy, an idea which proved successful in Halide [16] in terms of optimization and ease of use.

## III. Results

We modified MLIR to include dialects for the GSW and B/FV schemes, as well as custom lowerings and optimizations. We wrote lowering passes to and from the two FHE dialects and fully integrated them into the MLIR compiler infrastructure. A piece of code already lowered into the MLIR

Standard Dialect can be lowered through the GSW or B/FV schemes completely into the LLVM dialect by using the passes mentioned in the caption to Figure 3.

## IV. Conclusion and Future Work

We created SyFER-MLIR, an extensible, modular, optimizing compiler for encrypted programs. SyFER-MLIR automatically generates homomorphically encrypted code from any language which compiles to MLIR to any target architecture supported by LLVM. Currently, the GSW and B/FV schemes are supported. There are still many improvements to be made, such as the introduction of more FHE schemes like the CKKS, BGV, and the RNS variant of B/FV. More optimization passes can be written within the GSW and B/FV dialects themselves. Additionally, conversions can be written from more existing dialects into the FHE dialects to optimize the lowering process. Finally, SyFER-MLIR could be modified to support parallelization / multithreading to significantly increase its performance.

## V. Acknowledgements

The authors would like to thank Srinivasan and Deepa Govindarajan for their constant support. The authors would also like to thank Charles Leiserson, Srini Devadas, and Slava Gerovitch for their invaluable feedback and advice.

This research was supported in part by the MIT PRIMES program, and in part by LANL grant 531711. William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DE-SC0019323. Research was sponsored by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

## References

[1] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (CSUR)*, 51(4):1–35, 2018.

[2] D. Archer, L. Chen, J. Hee Cheon, R. Gilad-Bachrach, R. A. Hallman, Z. Huang, X. Jiang, R. Kumaresan, B. A. Malin, H. Sofia, Y. Song, and S. Wang. Applications of homomorphic encryption. Technical report, Microsoft Research, Redmond, WA, 2017.

[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.

[4] H. Chen, K. Laine, and R. Player. Simple encrypted arithmetic library-seal v2. 1. In *International Conference on Financial Cryptography and Data Security*, pages 3–18. Springer, 2017.

[5] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.

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

[7] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012.

[8] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Annual Cryptology Conference*, pages 75–92. Springer, 2013.

[9] D. Gunning, A. Hannun, M. Ibrahim, B. Knott, L. van der Maaten, V. Reis, S. Sengupta, S. Venkataraman, and X. Zhou. Crypten: A new research tool for secure machine learning with pytorch, 2019.

[10] S. Halevi, Y. Polyakov, and V. Shoup. An improved rns variant of the bfv homomorphic encryption scheme. In *Cryptographers' Track at the RSA Conference*, pages 83–105. Springer, 2019.

[11] S. Halevi and V. Shoup. Design and implementation of a homomorphic-encryption library, 2013.

[12] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[13] C. Lattner, J. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[14] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[15] PALISADE Lattice Cryptography Library (release 1.9.2). https://palisade-crypto.org/, April 2020.

[16] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, Dec. 2017.

[17] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.

[18] S. S. Sathya, P. Vepakomma, R. Raskar, R. Ramachandra, and S. Bhattacharya. A review of homomorphic encryption libraries for secure computation. *arXiv preprint arXiv:1812.02428*, 2018.