HECO: Fully Homomorphic Encryption Compiler

Alexander Viand, Patrick Jattke, Miro Haller, Anwar Hithnawi

ETH Zurich

Abstract

In recent years, Fully Homomorphic Encryption (FHE) has undergone several breakthroughs and advancements leading to a leap in performance. Today, performance is no longer a major barrier to adoption. Instead, it is the complexity of developing an *efficient* FHE application that currently limits deploying FHE in practice and at scale. Several FHE compilers have emerged recently to ease FHE development. However, none of these answer how to automatically transform imperative programs to secure and efficient FHE implementations. This is a fundamental issue that needs to be addressed before we can realistically expect broader use of FHE. Automating these transformations is challenging because the restrictive set of operations in FHE and their non-intuitive performance characteristics require programs to be drastically transformed to achieve efficiency. Moreover, existing tools are monolithic and focus on individual optimizations. Therefore, they fail to fully address the needs of end-to-end FHE development. In this paper, we present HECO, a new end-to-end design for FHE compilers that takes high-level imperative programs and emits efficient and secure FHE implementations. In our design, we take a broader view of FHE development, extending the scope of optimizations beyond the cryptographic challenges existing tools focus on.

1 Introduction

Privacy and security are gaining tremendous importance across all organizations, as public perception has shifted and expectations, including regulatory demands, have increased. This has led to a surge in demand for secure and confidential computing solutions that protect data's confidentiality in transit, rest, and in-use. Fully Homomorphic Encryption (FHE) is a key secure computation technology that enables systems to preserve the confidentiality of data at any phase; hence, allowing outsourcing of computations without having to grant access to the data. In the last decade, theoretical breakthroughs propelled FHE to a practical solution for a wide range of applications [1–3] in real-world scenarios. In addition, end-user facing deployments have started to appear, e.g., in Microsoft Edge's password monitor [1]. With upcoming hardware accelerators for FHE promising further speedup [4,5], FHE will soon be competitive for an even wider set of applications.

Though promising in its potential, developing *efficient* FHE applications remains a complex and tedious process that even experts struggle with. A large part of this complexity arises from the need to map applications to the unique programming paradigms imposed by FHE (cf. § 2.3). Properly optimized code for this paradigm is often several orders of magnitude more efficient than poorly adapted code, making optimization essential for practical FHE applications and posing a major barrier to wider adoption.

Fully Homomorphic Encryption is a nascent field and still actively evolving, with ongoing research on the cryptography, software implementations, and, increasingly, on hardware accelerators. As a result, tools must be designed to accommodate and adapt to to this fast moving field. Existing compilers (cf. § 7), however, are mostly rigid, monolithic tools with a narrow focus on individual sub-optimizations. Whereas experts usually transform applications in ways that accelerate them by orders of magnitude, existing tools have mostly focused on smaller-scale optimizations that result in small constant-factor speedups. While these represent important contributions, they are insufficient to make the kind of qualitative performance difference that is necessary to achieve practical FHE. In order to overcome these limitations, we need to fundamentally *rethink the architecture of FHE compilers* and *develop novel optimizations* that abstract away the complexity FHE and address the limitations of existing tools. In this paper, we present HECO, a new multi-stage optimizing FHE compiler. Our architecture provides, for the first time, a true end-to-end toolchain for FHE development. In addition, we propose novel transformations and optimizations that map imperative programs to the unique programming model of FHE.

E2E Architecture. Expert developers naturally structure the development of FHE applications into different stages. First, they consider how to efficiently map applications to the unique paradigm of FHE, e.g., how to minimize the need for data movement. Only afterward do they consider lower-level issues, such as fine-tuning the program to exploit scheme-specific optimizations. Currently, most developers then target software libraries such as Microsoft SEAL [6] which realize the underlying FHE schemes and implement a range of cryptographic optimizations. However, with more focus on hardware acceleration, there is an emerging need to optimize for individual hardware targets, which libraries are usually ill-suited to accomplish. Based on this progression of abstraction levels, we identified four phases of converting an application to an efficient FHE implementation: *program transformation*, *circuit optimization*, *cryptographic optimization*, and *target optimization* (cf. § 3.1).

Compilers need to be able to accommodate a wide variety of optimizations across these different levels of abstraction. However, existing compilers usually abstract FHE computations as circuits consisting of the basic homomorphic operations and scheme-specific operations for managing noise. This is a natural representation since FHE computations are mathematically modeled as arithmetic circuits. However, many optimizations at the high-level (program transformation) or at the lowest level (target optimization) cannot be fully expressed in the circuit setting because they consider aspects such as data flow or memory management which have no natural correspondence in this abstraction. We instead propose a set of Intermediate Representations (IRs) based on the requirements of each phase that allow us to naturally and efficiently express optimizations at these different levels. We realize these IRs using the MLIR compiler framework [7], which provides a standardized way to define and operate on domain-specific IRs. MLIR enables the transfer of optimizations between different projects, including across domains, and provides a powerful software framework. Specifically, it is well suited to represent and optimize high-level programs in a way that circuit-based tools are not.

Automated Mapping to Efficient FHE. HECO supports the automatic transformation of high-level programs to FHE's unique programming paradigm. Experts spend significant time considering *how* to best express an application in the FHE paradigm, only considering the other aspects once the program is *efficiently* expressible using native FHE operations. Existing tools, in contrast, typically disregard this arguably most important phase of the FHE development process.

In HECO, developers can express their algorithms conveniently in the standard imperative paradigm, e.g., using loops that access/modify individual vector elements. However, such programs do not align well with the restricted set of operations offered by FHE schemes, requiring the compiler to translate and restructure the application. We focus on transformations targeting the Single Instruction, Multiple Data (SIMD) parallelism of most modern schemes, which allows one to *batch* many (usually, $2^{13}-2^{16}$) different values into a single ciphertext and compute over all simultaneously. Batching is used by experts to drastically reduce ciphertext expansion and computational overhead, frequently improving runtimes by several orders of magnitude when compared to naive implementations.

While it is arguably the single most important optimization for many applications, unlocking its performance potential currently requires significant expertise and experience in writing FHE applications. Because these schemes do not offer the data movements operations (e.g., scatter/gather/permute) usually present in the context of vector operations, existing approaches from the traditional compiler literature do not translate well. Specifically, FHE only natively supports element-wise SIMD operations and cyclical rotations of the elements inside a ciphertext and existing algorithms must be transformed dramatically in order to be expressed solely from these operations. We devise a series of transformations and optimizations that can translate batching-amenable programs to fully exploit SIMD operations while minimizing the need for data movement (c.f. § 4).

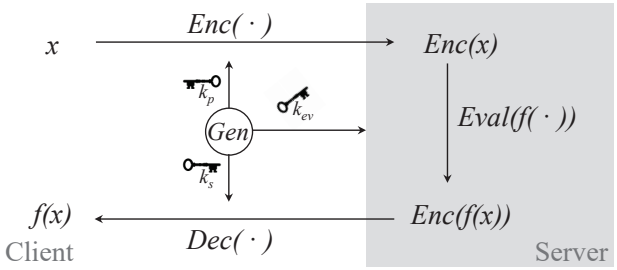


Figure 1: Using FHE, a third party can compute on encrypted values without requiring access to the underlying data.

In our evaluation, we show that HECO can match the performance of expert implementations, providing up to 3500x speedup over naive non-batched implementations (c.f. § 6). We open-source HECO and we hope that it will help to advance the FHE development ecosystem. HECO decouples optimizations from front- and back-end logic, allowing it to be easily extended to different languages, FHE libraries, accelerators, and novel optimizations as they emerge.

2 Background

In this section, we briefly introduce the notion of FHE and key aspects of modern FHE schemes.

2.1 Fully Homomorphic Encryption

In a *homomorphic* encryption (HE) scheme, there exists a homomorphism between plaintext and ciphertext operations such that, e.g., $\mathrm{Dec}(\mathrm{Enc}(x+y)) = \mathrm{Dec}(\mathrm{Enc}(x) \oplus \mathrm{Enc}(y))$ in the case of additively homomorphic encryption. While *additively* and *multiplicative* HE schemes (e.g., Paillier [8] and textbook RSA [9], respectively) have been known for many decades, *fully* homomorphic encryption (FHE) schemes that support an arbitrary combination of both operations remained practically infeasible until Gentry's breakthrough in 2009 [10]. This allows a third party to compute on encrypted data, without requiring access to the underlying data. Specifically, FHE is traditionally defined for *outsourced computation*, where a client provides encrypted data x and a function f to a server which computes and returns $f(x)$ as shown in Figure 1. The client can decrypt the returned result while the server learns nothing about inputs, intermediate values, or results. Beyond this simple setting, the function (and additional inputs) can also be supplied by the server, opening up additional interesting deployment scenarios. For example, Private Set Intersection (PSI) as used in Microsoft Edge's password monitor [1], where the client sends encrypted login credentials to

the server, which compares them against its own database of leaked usernames and passwords. In the decade since its first realization, FHE performance has improved dramatically: from around half an hour to compute a single multiplication to only a few milliseconds. Nevertheless, this is still seven orders of magnitude slower than a standard, signed multiplication executed on a modern CPU. However, this gap is expected to be reduced significantly with the emergence of dedicated FHE hardware accelerators [5, 11–13].

2.2 FHE Schemes

We briefly describe the BFV scheme [14, 15] as a representative of the largest family of FHE schemes. We focus on aspects relevant to FHE application development and refer to the original papers for further details. Since most modern FHE schemes follow a similar pattern, the descriptions also mostly apply to the BGV and CKKS schemes [16, 17].

In BFV, plaintexts are polynomials of degree n (usually $n > 2^{12}$) with coefficients modulo q (usually $q > 2^{60}$). Encryption introduces *noise* into the ciphertext, which is initially small enough to be rounded away during decryption but accumulates during homomorphic operations. As basic operations, BFV supports additions and multiplications over ciphertexts. Additions increase the noise negligibly, but multiplications affect it significantly. Managing noise is crucial to prevent ciphertext corruption, which manifests as a failing decryption. The noise limits computations to a (parameter-dependent) number of consecutive multiplications (multiplicative *depth*) before decryption fails. While *boostrapping* can reduce the noise homomorphically, it introduces significant overheads and must therefore be eliminated or minimized to achieve practical FHE solutions.

Using the Chinese Remainder Theorem (CRT), it is possible to *encode* a vector of n integers into a single polynomial, with addition and multiplication acting slot-wise (SIMD). Since the polynomial of degree n is usually between 2^{13} and 2^{16} for security, it can significantly reduce ciphertext expansion and computation cost. BFV also supports rotation operations over such *batched* ciphertexts, which cyclically rotate the vector's elements. Finally, BFV includes a variety of noise-management (or *ciphertext maintenance*) operations, which do not change the encrypted message but can reduce noise growth during computations.

Security. Modern FHE schemes rely on post-quantum hardness assumptions, widely believed to be secure for the foreseeable future. The community has developed estimates of their concrete hardness [18] and parameter choices for several FHE schemes have been standardized [19]. Nevertheless, some attention must be paid to security when using FHE. For example, FHE does not provide *integrity* by default, i.e., a server might perform a different calculation than requested or none at all. There exist techniques to address that, ranging from zero-knowledge-proofs to hardware attestation [20]. Additionally, FHE does not provide by default *circuit privacy*, i.e., a client might be able to learn information about the applied circuit. Different techniques, varying in practicality and protection level, can be used to address this [21, 22]. Finally, issues can appear when using *approximate* homomorphic encryption (e.g., CKKS), with attacks that can recover the secret key from the noise embedded in ciphertext decryptions [23]. Recent work has shown how adding differentially private noise can mitigate these attacks [24], but some concerns remain.

2.3 FHE Programming Paradigm

FHE imposes a variety of restrictions on developing programs: some derive from the definition of FHE and its security guarantees, while others result from scheme restrictions and cost models. For example, FHE's security guarantees make it necessarily data-independent, hence preventing branching based on secret inputs. While some forms of branching can be *emulated*, all branches must be evaluated, resulting in a potentially significant degradation of performance. In addition, FHE schemes only offer a limited set of data types and operations, with addition and multiplication as basic operations. Applied over binary plaintext spaces ($\mathbb{Z}_2$), this technically enables arbitrary computation. However, the best performance is usually achieved with larger plaintext spaces (e.g., $\mathbb{Z}_t$ for $t \gg 2$). In this setting, computations are equivalent to arithmetic circuits, which can only compute polynomial functions. Non-polynomial functions can be approximated, but this is typically prohibitively inefficient. While recent works have explored homomorphic conversions between binary and arithmetic settings [25, 26] and introduced *programmable bootstrapping* to approximate non-polynomial functions [27], these approaches are not yet practical enough for widespread adoption.

As a result, developing FHE applications requires fundamentally rethinking how programs are written. Generally, developers need to rethink their approach, e.g., using branch-free algorithms well-suited to low-degree polynomial approximations. In addition, the large size of FHE ciphertexts, which is required for security reasons, is a significant source of both communication and computation overhead. However, it also presents an opportunity, as many[1] schemes support *batching*, which allows encrypting many values into the same ciphertext. This reduces ciphertext expansion and enables element-wise operations in a Single Instruction, Multiple Data (SIMD) fashion. Data movement in FHE is incredibly restricted, affording no efficient ways to permute the batched data after encryption, with the exception of cyclical rotations. As a result, efficient FHE algorithms are usually drastically different from their plaintext equivalents. Adapting to this unique programming paradigm requires a lot of experience and poses a significant barrier to entry for non-experts.

[1] Specifically, schemes from the Ring-LWE family that B/FV, BGV and CKKS belong to.

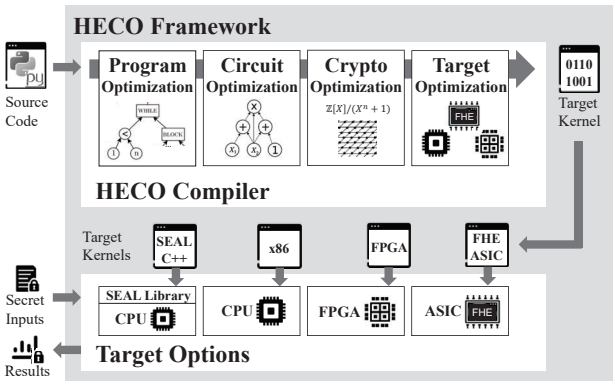


Figure 2: Overview of our end-to-end design, showing the compilation flow from a high-level input program to an efficient FHE kernel running on a target backend.

3 End-to-End FHE Compiler Design

This section first provides a system overview of HECO and then presents its core components, beginning with the overall framework design, followed by a discussion of our compiler architecture, and finally, give an overview of the transformations and optimizations that constitute the compilation pipeline.

3.1 System Overview

HECO proposes a multi-staged approach to FHE compilation that encompasses: *(i) Program Transformations*, which restructure high-level programs to be efficiently expressible using native FHE operations, *(ii) Circuit Optimizations*, which primarily focuses on changes that reduce noise growth in the FHE computation, *(iii) Cryptographic Optimizations*, which instantiate the underlying scheme as efficiently as possible for the given program, and *(iv) Target Optimizations*, which map the computation to the capabilities of the target. We propose a set of Intermediate Representations (IRs) designed to provide a suitable abstraction of each stage, allowing us to naturally and efficiently express optimizations at these different levels. In contrast, existing compilers usually abstract FHE computations as circuits[2] which does not allow them to fully express many optimizations at the high-level (program transformation) or at the lowest level (target optimization) because these need to consider aspects such as data-flow or memory management which have no natural correspondence in a circuit representation. In HECO, high-level programs are lowered through a series of transformations, using multiple increasingly lower-level IRs to produce the target kernel. These kernels can then be targeted and run against various back-end options. We provide a user-facing Python framework that abstracts away the complexities of this process, supports a Python-embedded Domain Specific Language for FHE, and

[2]This is natural since FHE computations are usually modeled mathematically as arithmetic circuits.

```python
def server(x_enc, y_enc, public_context):
  p = FrontendProgram()
    with CodeContext(p):
      def euclidean_sq(x: Tensor[8, Secret[int]],
                       y: Tensor[8, Secret[int]])
                      -> Secret[int]:
        sum: Secret[int] = 0
        for i in range(8):
          d = x[i] - y[i]
          sum = sum + (d * d)
        return sum

  # compile FHE code
  f = p.compile(context=public_context)
  # run FHE code using SEAL
  r_enc = f(x_enc, y_enc)
  return r_enc
```

Listing 1: Example server-side code using HECO.

```python
def client(x : Tensor[int], y : Tensor[int]):
    # Select SEAL backend, scheme and params
    context = SEAL.BFV.new(poly_mod_degree=2048)

    # encrypt input
    x_enc = context.encrypt(x)
    y_enc = context.encrypt(y)

    # send enc input to server
    r_enc = server(x_enc, y_enc, context.pub())
    result = context.decrypt(r_enc, context)
```

Listing 2: Corresponding client-side code, outsourcing the computation of the (squared) euclidean distance.

provides a unified experience for development, compilation and execution. We provide an overview of our end-to-end design in Figure 2. In the remainder of this section, we describe HECO's components, abstractions, and compilation stages.

3.2 HECO Framework

HECO's framework ties together the front end, compiler, and the various back ends into a unified development experience. It allows developers to edit, compile and deploy their applications from a familiar Python environment. In order to provide an intuition of the developer experience in our system, we provide an example of using HECO to compile and run an FHE program in Listing 1 (Server) and Listing 2 (Client). By wrapping FHE functions in `with` blocks, we can operate on them as first-class entities, making compilation explicit. Our framework provides the necessary infrastructure to run programs directly from the front end, allowing developers to integrate FHE functionality into larger applications easily.

Python-Embedded DSL. HECO uses Python to host its Domain-Specific Language (DSL), inheriting Python's syntax and general semantics. We want to allow developers to

write programs in as natural a fashion as possible, and merely require type annotations to denote inputs that are `Secret`. In order to facilitate this, HECO supports (statically sized) loops, access to vector elements, and many other high-level features that do not have a direct correspondence in FHE. Since our compilation approach requires a high-level representation of the input program, including these non-native operations and the control-flow structure, we cannot follow the approach used by most existing tools. These tend to execute the program using placeholder objects that record operations performed on them, which is equivalent if considering FHE programs as circuits but removes most of the high-level information about the program structure. Instead, we use Python's extensive introspection features to parse the input program and translate the resulting Abstract Syntax Tree (AST) directly to our high-level IR.

3.3 Compiler Infrastructure

The core of HECO is an optimizing compiler that translates and optimizes programs by lowering them through a series of progressively lower-level Intermediate Representations (IRs). This section describes how we build upon the MLIR framework to realize HECO's compiler design.

Multi-Level Intermediate Representations. HECO's middle end exposes multiple levels of abstractions to facilitate our multi-stage compilation & optimization approach. This is realized through a series of Intermediate Representations (IRs), as seen in Figure 3. We leverage the MLIR framework [7], which was designed specifically to facilitate *progressive lowering*, introducing additional IRs to reduce the complexity of each lowering step. MLIR defines a common syntax for IR operations, for example, an addition might be represented as `%2 = artih.addi(%0, %1) : (i16, i16) -> i16`. MLIR is strongly typed, however, for conciseness, we will omit the details of type conversions when discussing transformations. Intermediate Representations in MLIR are composed of sets of operations known as *dialects*. We define a custom dialect for our high-level abstraction of FHE (`heco::fhe`) and combine this with built-in dialects for vector operations (`mlir::tensor`), plaintext arithmetic (`mlir::arithmetic`) and basic program structure (`mlir::affine`, `mlir::func`) to realize our High-Level Intermediate Representation (HIR). In addition, we define dialects for each of the supported FHE schemes, mirroring their natively supported operations (`heco::bfv`, `heco::bgv`, `heco::ckks`). MLIR also includes a variety of standard simplification passes, which can be extended to custom dialects by defining appropriate interfaces.

Supporting Different Back-Ends. FHE is actively evolving, and as such, tools need to be able to adapt to new and improved implementations, both in software and hardware. This requires a high level of modularity and flexibility from the compiler. In HECO, we achieve this by using target-specific dialects, which can be customized and extended as new back ends are introduced. While traditional library-based implementations targeting CPUs and GPUs share a common API (conceptually, if not technically), upcoming FPGA and ASIC accelerators for FHE [4, 28–30]) feature a much lower-level interface. These systems are designed to efficiently realize the required mathematical operations in the modular rings of polynomials that underly most FHE schemes, and as a result, their Instruction Set Architectures (ISA) operate on this level. In order to support this, HECO is designed to be easily extended to match this abstraction level, featuring MLIR dialects for both bignum polynomial ring operations (`heco::poly`) and for the commonly used Residue Number System (RNS) approach using the Chinese Remainder Theorem (CRT) to split these large datatypes into hardware-sized elements (`heco::rns`). In addition to targeting hardware accelerators, the ability to lower to this level also allows targeting `x86` directly via LLVM IR and the LLVM toolchain.

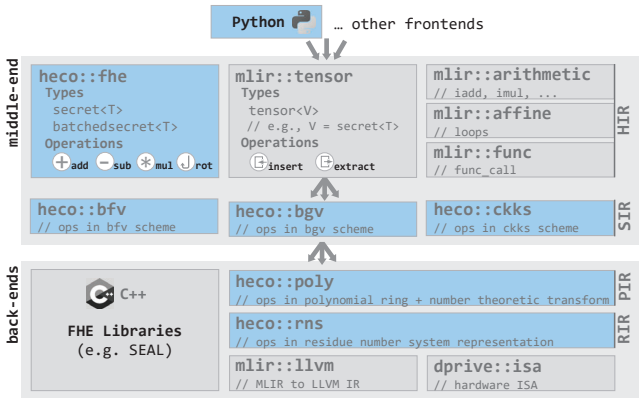


Figure 3: Overview of HECO's dialects, which define the operations used in the Intermediate Representations (IRs).

3.4 Transformation & Optimization

The transformations and optimizations in HECO are grouped according to the four stages of compilation we identified, and we present them accordingly in the following.

Program Transformations. The first phase of compilation focuses on high-level transformations and optimizations. This includes a wide variety of general (e.g., constant folding, common sub-expression elimination) and FHE-specific optimizations that allow developers to write code more naturally by removing the need for menial hand-optimization. Most importantly, however, it focuses on optimizations that map the input program to FHE's unique programming paradigm, such as the automated batching optimizations, which we present in more detail in § 4. Previous work has shown that performance

differences between the runtimes of well-mapped and naively-mapped implementations can easily reach several orders of magnitude [31]. As a result, a significant part of our focus in HECO is on this level of abstraction, which existing FHE tools generally do not support.

Circuit Optimizations. After mapping to the FHE paradigm, the program is conceptually equivalent to an arithmetic circuit of native FHE operations. This is the level of abstraction considered by the vast majority of existing tools. Optimizations at this stage are mostly concerned with managing the noise growth in the computation. For example, a variety of optimizations that try to re-arrange the arithmetic operations to reduce the number of sequential multiplications have been proposed [32–35]. However, even state-of-the-art optimizations in this style are likely to accelerate a program by only around 2x in practice [31]. We omit a detailed description of these techniques here as they are not the focus of this paper. More importantly, this level of abstraction is also where we must consider ciphertext maintenance operations. These do not modify the encrypted messages, but significantly affect future noise growth, making them essential for practical FHE. HECO uses a traditional approach of inserting relinearization operations [15] between all consecutive multiplications. This is always correct but not necessarily strictly optimal, and more sophisticated strategies [36–38] could offer further improvements. The modularity of our design makes it a straightforward future work to include these techniques, but since these have been explored in the past, we do not focus on them in this paper.

Cryptographic Optimizations. In the third phase, we consider *cryptographic optimizations* focused on instantiating the underlying FHE scheme as efficiently as possible. When targeting existing FHE libraries, the primary challenge is parameter selection: identifying the smallest (i.e., most efficient) parameters that still provide sufficient noise capacity to perform the computation correctly. Different techniques have been proposed to estimate the expected noise growth of an FHE program [39–41]. These include theoretical noise analysis, where recent work has achieved tighter bounds for some schemes [41], but which generally tend to significantly overestimate noise growth [42], leading to unnecessarily large parameter choices. As a result, experts primarily still rely on a trial-and-error process to experimentally determine the point at which noise invalidates the results. HECO includes basic automatic parameter selection based on a simple multi-depth heuristic but also allows experts to easily override these suggestions. When targeting hardware directly, rather than through libraries, further optimization opportunities open up. For example, many ciphertext maintenance operations can be instantiated in different ways, offering trade-offs between runtime, memory consumption, and noise behavior. While libraries tend to implement a general-purpose compromise, compilers can adaptively choose the most appropriate approach for a given computation. However, this requires re-expressing the complex underlying logic of FHE schemes inside the compiler. HECO inherits a powerful system of abstractions and optimizations for computationally intense mathematics from the MLIR framework, allowing our system to be easily extended with such optimizations in the future.

Target Optimization. Finally, in the fourth phase, we consider *target-specific optimizations*. In addition to general code generation optimizations, there is a significant opportunity for FHE-specific optimizations at this level. For example, when available, FHE benefits greatly from instruction set extensions such as `AVX512`. This concept has already been explored in the context of libraries [43], and implementing similar techniques in HECO should be straightforward given our modular design. When targeting hardware, FHE accelerators impose non-trivial constraints on memory and register usage, due to complex memory hierarchies. Initial work in this space has already shown that code generation and scheduling can have a significant impact on accelerator performance [29,30]. HECO supports optimizations at this level through our low-level dialects for the underlying math, and can easily be extended with target-specific dialects for the Instruction Set Architectures (ISAs) of upcoming accelerators, e.g., those developed by the DARPA DPRIVE program [4].

4 Automatic SIMD Batching

In this section, we introduce our automated SIMD batching optimization, which is part of our program transformation stage and maps traditional imperative programs to the restrictive SIMD-like setting of state-of-the-art FHE schemes.

4.1 SIMD Batching

Effective use of batching is arguably the single most important optimization for many applications and is omnipresent in most state-of-the-art FHE results. Due to the large capacity of FHE ciphertexts (usually $2^{13} - 2^{16}$ slots), applying batching has the potential to drastically reduce ciphertext expansion overhead and computation time. While batching can be used to trivially increase throughput, most FHE applications are constrained by latency. However, employing batching effectively to improve performance on a single input is non-trivial due to the restrictions imposed by FHE's unusual programming paradigm. Therefore, unlocking the performance potential of batching currently requires significant expertise and experience in writing FHE applications. In the following, we present a simple example that showcases the drastic transformations that can be required to achieve efficient batching, followed by a brief introduction of a folklore technique that demonstrates common patterns of FHE batching optimizations.

Example Application – Image Processing. We consider a simple image processing application (see Listing 3a), which nevertheless features a complex loop nest structure and non-trivial index patterns. Specifically, we consider a Laplacian

```python
def foo(img: Tensor[N, Secret[f64]]):
  img_out = img.copy()
  w = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]
  for x in range(n): # loop over pixels
    for y in range(n):
      t = 0
      for j in range(-1, 2): # apply kernel
        for i in range(-1, 2):
          t += w[i+1][j+1] * img[((x+i)*n+(y+j))%N]
          img_out[(x*n+y)%N] = 2*img[(x*n+y)%N] - t
  return img_out
```

(a) Textbook implementation of a simple sharpening filter

```python
def foo_batched(img: BatchedSecret[f64]):
  r0 = img * -8
  r1 = img << -n-1
  r2 = img << -n
  r3 = img << -n+1
  r4 = img << -1
  r5 = img << 1
  r6 = img << n-1
  r7 = img << n
  r8 = img << n+1
  return 2*img-(r0+r1+r2+r3+r4+r5+r6+r7+r8)
```

(b) Optimized batched solution of the same program

Listing 3: Both of these functions apply a simple sharpening filter to an encrypted image of size $n \times n = N$, by convolving a 3×3 kernel (-8 in the center, 1 everywhere else) with the image. The version on the left encrypts each pixel individually, and follows the textbook version of the algorithm, operating over a vector of N ciphertexts. The version on the right batches all pixels into a single ciphertext and uses rotations (<<) and SIMD operations to compute the kernel over the entire image at the same time. Designing batched implementations requires out-of-the-box thinking in addition to significant expertise and experience.

Sharpening filter, i.e., a convolution of a ($3x3$) kernel over an image, implemented with wrap-around padding. The function is compatible with efficient arithmetic-circuit based FHE, as it does not use data-dependent branching and only requires homomorphic addition and multiplications operations. However, its current form makes use of nested loops accessing a complex set of indices, which is not very amenable to efficient batching as there appears to be little opportunity for operations over entire ciphertexts.

Nevertheless, there exists a significantly more efficient *batched* design, as seen in Listing 3b. In the optimized version, the input image is batched into a single ciphertext, and all homomorphic operations make full use of their SIMD nature. Instead of iterating the kernel over the image, nine copies of the image are made and each is rotated so that all elements interacting at a specific kernel position align at the same index. This is possible, because the relative offset between different pixels in the kernel remains static, even though the indices themselves are different for each iteration. The transformation enables the the runtime of the program to depend on the (small) kernel size, rather than the image size. As a result, the batched version is more than an order of magnitude more efficient than a naive implementation. These types of drastic transformations are common in state-of-the-art FHE applications and significant experience is required to develop an intuition for this unusual programming paradigm.

Rotate-and-Sum. In the example above, only interactions between values in different ciphertexts were required. However, it is also possible to efficiently realize certain operations on the elements of a single ciphertext; we now describe a common folklore technique used to achieve this: The *rotate-and-sum* algorithm allows us to efficiently sum up the elements of a ciphertext, using $(\log n)$ rotations (where n is the number of ciphertext slots). The algorithm proceeds by creating a copy of the current vector, rotating it and then adding both

Algorithm 1 Rotate-and-Sum

Algorithm SumVectorPowerTwo(x, n)
  for $i \leftarrow \frac{n}{2}$ **downto** 1 **by** $i \leftarrow \frac{i}{2}$
    $x \leftarrow x + \text{Rotate}(x, i)$
  return x

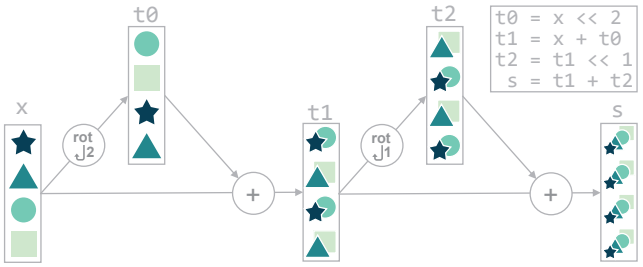


Figure 4: Illustration of how repeated copying and rotating can be used to compute the sum of all elements in a ciphertext in a logarithmic, rather than linear, number of steps.

before repeating the same procedure with a lower offset (c.f. Algorithm 1). This is visualized in Figure 4 for a vector size of four. While this technique is applicable, the performance benefits are usually overshadowed by more radical transformations, such as the example shown above. However, using rotate-and-sum and similar rotation-based approaches can be worthwhile if it enables other parts of the program to remain slot-aligned.

4.2 Automatic Batching Approach

Experts generally rely on their experience with the FHE programming paradigm to transform and optimize programs for batching, posing a high barrier to entry for non-expert developers. Instead, formal methods to automatically translate traditional imperative programs into efficient batched FHE solutions are required. We assume the input program computes

```
%1=tensor.extract %x[i]          %1=fhe.mul(%x, %m_i)
%2=tensor.extract %y[j]          %2=fhe.rotate(%1, -i)
%3=fhe.add(%1, %2)               %3=fhe.mul(%y, %m_j)
%4=tensor.insert(%3,%z[i])       %4=fhe.rotate(%3, -j)
                                 %5=fhe.add(%2, %4)
                                 %6=fhe.rotate(%5, i)
                                 %7=fhe.mul(%z, %mn_i)
                                 %8=fhe.add(%6, %7)
```

(a) Input Program (b) Naive Batching

Listing 4: Strawman batching approach for `z[i] = x[i] + y[j]`, showing the necessary rotations and multiplications with masking vectors: `%m_i`, `%m_j` are zero everywhere except at i or j, respectively; `%mn_i` is one everywhere except at i.

the elements from vectors of secret values in a non-SIMD fashion (e.g., Listing 3a). Of course, this can be naively realized by encrypting each vector element into one ciphertext , but this usually does not achieve acceptable performance due to the high overhead of FHE. The goal of automated batching is to amortize the cost of each FHE operation by utilizing as many ciphertext slots as possible for meaningful computation. In the following, we discuss two potential alternative approaches and their drawbacks before introducing our approach.

Strawman Approach. Batching each vector in the input program into a ciphertext will trivially achieve a 'batched' solution. However, this raises the question of how to execute the computations over individual elements present in the program. Element-wise access (`extract` and `insert`) are not native FHE operations and must instead be emulated, requiring several rotations and ciphertext-plaintext multiplications. For example, Listing 4 shows how `x[i] = x[i] + y[j]` can be emulated in the batched setting. However, this replaces each FHE operation from the naive, vector-of-ciphertexts approach, with multiple expensive FHE operations. As a result, unless ciphertext expansion is significantly more important than runtime, this approach is virtually always ill-advised. Therefore, most existing FHE tools use the vector-of-ciphertexts approach rather than attempting to perform batching.

Alternative Approaches. There have been initial attempts at performing automated batching for FHE using Synthesis-based approaches [44]. While these can, in theory, achieve the drastic transformations required to exploit batching, they are not suitable for practical use in real-world code development, as they do not scale beyond toy-sized program snippets, and even those can take minutes to optimize. Alternatively, one might consider applying traditional Superword-Level Parallelism (SLP) vectorization algorithms [45–47], as these try to group operations into SIMD instructions. However, these generally rely on the ability to efficiently scatter/gather elements into and out of vectors, which is only possible at a high cost in FHE. While some recent work can reason about the cost of data movement, it does not consider how data movement introduced at the beginning of the program might affect later parts of the program [47].

HECO's Approach. HECO's batching transformation starts with the core idea of the strawman approach, i.e., batching vectors of secrets into ciphertexts but eliminates the overhead of emulating `insert`/`extract` operations for batching amenable programs. Rather than directly emulating these operations, we instead translate the homomorphic operations in which they appear as operands. While this still requires inserting rotation operations, it allows operations with compatible index access patterns to be mapped to the same emulated code if using an appropriate algorithm. As a result, a simple built-in simplification pass can eliminate the duplicates. In the case of well-structured programs, this can completely eliminate emulation related code. For example, for the program from the previous section (Listing 3a), HECO produces exactly the optimized code seen in Listing 3b, which does not contain any emulation-related code.

HECO Batching Pipeline. HECO's batching transformation is composed of a series of smaller passes, each interleaved with built-in simplification passes. Before the main batching passes, we first perform a series of preprocessing steps that unroll statically-sized loops, merge sequential associative binary operations into n-ary group operations, and perform a type conversion from vectors of secrets to `BatchedSecrets`, which are HECO's high-level abstraction of ciphertexts. Following this, the main pass walks through the program and transforms each operation over secret vector elements into operations over entire vectors. Similar to the strawman approach, this involves introducing rotations, but our approach does not require multiplications with masks. Additionally, the way we perform these translations allows us to 'chain' them so that consecutive operations on the same vector elements do not result in separate emulation code. After the main pass and the associated simplification pass, we apply the rotate-and-sum technique where applicable, which is enabled by both the merging of operations during the preprocessing phase and the exposure of same-ciphertext operations by the main pass. In the following, we first outline some key preprocessing steps before explaining the two main optimization steps.

4.3 Preprocessing

In addition to standard simplifications and canonicalization of the IR (i.e., bringing operations into a standardized 'canonical' form to reduce the complexity of the IR), we also apply two more specialized transformations.

Merging Arithmetic Operations. During the preprocessing stage, we combine chained applications of (associative and

commutative) binary operations into larger arithmetic operations with multiple operands (e.g., merging `x=a+b; y=x+c` to `y=a+b+c`). This removes chains of dependencies and replaces them with a single operation, making it easier to identify *rotate-and-sum* optimization opportunities. In addition, it can also allow more efficient direct lowerings, such as when performing the product over n elements: which can be lowered efficiently to a multiplication tree with depth $\log n$.

Type Conversion. While tensors can be arbitrarily (re)shaped, all FHE ciphertexts used in a homomorphic computation must have compatible parameters, which implies a fixed number of slots. Therefore, we perform type conversion, converting all vector operations over secret values to operations over the `BatchedSecret` type, an abstract representation of ciphertexts. We convert multi-dimensional tensors to vectors using column-major encoding and scale up any secret vector operands to the size of the largest (secret) vector present, padding the plaintext as necessary. This does not impact the result of the computation, as the existing code will never access these additional elements.

4.4 Automatic SIMD-fication

This pass replaces scalar operations over vector elements (e.g., `x[i] + y[j]`) with SIMD operations, applying the same operation to each element of the ciphertext. At its core, the pass is a linear walk over all (arithmetic) homomorphic operations in the program, as seen in Algorithm 2. For each operation, we *(i)* identify in which ciphertext slot the result should be computed *(ii)* transform the operands so that they are suitable for such a SIMD-operation, and *(iii)* insert `extract` operations in situations where a scalar is expected (including uses in later operations that have yet to be transformed). Note that, by itself, this transformation does not actually remove any code. However, it will expose common patterns (e.g., such as those occurring in loops) and cause operations over *compatible* indices to be translated to the same SIMD operations. This allows the following clean-up pass to remove these now-redundant operations, which frequently includes duplicate arithmetic operations and many or all of the operations inserted to ensure consistency.

Target Slot Selection. When translating an operation with operands corresponding to different vector positions (e.g., `x[i]+y[j]`), we must bring the elements of interest into alignment by issuing a rotation for at least some of the operands. However, there are usually multiple valid solutions (e.g., `rot(x, j-i)` vs. `rot(y, i-j)`), especially for operations with multiple operands, which occur frequently as a result of our preprocessing stage An obvious approach would be to rotate each operand (e.g., `x[i]`) so that the element of interest is moved to slot 0. Then, performing the SIMD operation over the rotated operands produces the result in the same slot.

Algorithm 2 Batching Pass

```
1:  Algorithm BatchPass(G)
2:      V, E ← G
3:      foreach op ∈ V ∧ type(op) = fhe.secret:
4:          ts ← SelectTargetSlot(op, V, E)
5:          OperandConversion(op, ts, V, E)
6:          foreach v ∈ V ∧ (op, v) ∈ E:
7:              u ← fhe.extract[v, ts]
8:              Replace(v, u, V, E)
9:  procedure SelectTargetSlot(op, V, E)
10:     foreach v ∈ V ∧ (op, v) ∈ E:
11:         switch v:
12:             case fhe.insert[_, i]: return i
13:             case func.return: return 0
14:     foreach v ∈ V ∧ (v, op) ∈ E:
15:         switch o:
16:             case fhe.extract[_, i]:
17:                 return i
18:     return ⊥
19: procedure OperandConversion(op, ts, V, E)
20:     foreach v ∈ V ∧ (v, op) ∈ E ∧ type(v) = fhe.secret:
21:         switch v:
22:             case fhe.extract(x, i):
23:                 u ← fhe.rotate(x, i − ts)
24:                 Replace(v, u, V, E)
25:             case fhe.ptxt[p]:
26:                 p′ ← Repeat(p)
27:                 u ← fhe.ptxt(p′)
28:                 Replace(v, u, V, E)
29: procedure Replace(v, u, V, E)
30:     V ← (V \ {v}) ⋃ {u}
31:     foreach w ∈ V ∧ (v, w) ∈ E:
32:         E ← (E \ {(v, w)}) ⋃ {(u, w)}
33:     foreach w ∈ V ∧ (w, v) ∈ E:
34:         E ← (E \ {(w, v)}) ⋃ {(w, u)}
```

While this approach is straightforward and correct, it is unsuitable for an optimizing compiler, as it does not set the program up for further simplification. For example, in the case of a loop `for i in 0..10: z[i]=x[i]+y[i]`), each iteration would result in a unique operation with distinct operands, each rotated by different amounts.

Instead, HECO introduces the notion of a *target slot*, determined by the further uses of the result. For example, if the result of a computation is assigned to `z[k]`, we select k as the target slot, eliminating the rotation required afterward. If no clear target slot can be derived from the uses of the result, we use one of the operand indices as the target slot, removing the need to rotate that operand. Selecting the target slot this way reduces the immediate number of rotation operations created. More importantly, it reliably maps operations with the same *relative* index access patterns to the same set of ro-

tations and SIMD operations, allowing them to be eliminated by the following simplification pass. Since this approach is based purely on index access patterns, it works equally well for complex loop nests and heavily interleaved code.

Operand Conversion. In order to convert a scalar operation to a fully-batched SIMD operation, all non-batched inputs must be converted, as described in Algorithm 2 (OPERAND-CONVERSION). Note that, due to the linear-walk nature of the pass, all previous FHE operations have already been converted to fully-batched operations. As a result, any operand of type `fhe.secret` must be the result of an `fhe.extract` operation. This invariant allows us to 'chain' batched operations together by replacing the extraction-based operand with a rotation-based operand. Specifically, an operand extracted from slot i of vector x, is replaced with a rotation of x by $i - ts$, where ts is the target slot determined before.

Ensuring Consistency. We maintain the consistency and correctness of the program at each step of the optimization. Towards this, we first construct the new rotations and batched operations as additions to the program. We only replace occurrences of the old operation with the optimized version after we replace uses of the old operation with an `extract` operation that extracts the target slot of the new batched result. This ensures that, even if no further batching opportunities are found, the program remains correct. Since batched FHE schemes do not support true scalar values, we simply interpret scalars as ciphertexts where only slot 0 contains valid data. With this convention, any remaining extractions will eventually be converted to a rotation by $-ts$. However, in practice, this is rare as most of the `extract` operations we insert will in turn be converted to rotations when the next homomorphic operation is processed. As a result, these consistency-related `extract` operations are frequently eliminated completely at the end of the batching pass.

4.5 Rotate-and-Sum Pass

After the main pass and the associated simplification pass, we apply the *rotate-and-sum* technique where applicable. Since this optimization requires a holistic view of the operation, this would be significant if we did not merge sequential operations during preprocessing. While we used a sum over all elements when explaining the technique in the previous section, the technique can be generalized to any subset with a consistent stride. Additionally, it can also be used to compute products rather than sums. When applied to multiplication, it additionally has the benefit of automatically reducing the multiplicative depth of the expression as a side-effect.

Note that the pre-processing combination of binary operations into larger operations *must* happen before the main pass described above, as that pass would otherwise insert rotations between the different operations, making them no longer directly chained. The actual translation to a series of rotations and native binary operations, meanwhile, has to be performed *after* that pass, since it requires the operands to be entire ciphertexts, rather than scalars. Additionally, the de-duplication simplifications that can take place after the batching transformation can widen the applicability of this transformation by reducing the number of distinct ciphertexts appearing in the program.

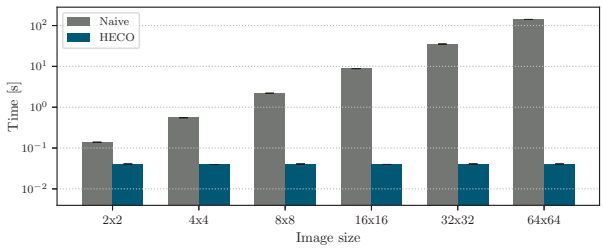
5 Implementation

We build HECO on top of the open-source MLIR framework [7], which is rapidly establishing itself as the go-to tool for domain-specific compilers and opening up the possibility of exchanging ideas and optimizations even beyond the FHE community. HECO consists of roughly 15k LOC of C++, with around 2k LOC of Python for the Python front-end. HECO uses the Microsoft Simple Encrypted Arithmetic Library (SEAL) as its FHE backend. SEAL, first released in 2015, is an open-source FHE library implemented in C++ that is thread-safe and heavily multi-threaded itself. SEAL implements the BFV, BGV and CKKS schemes.

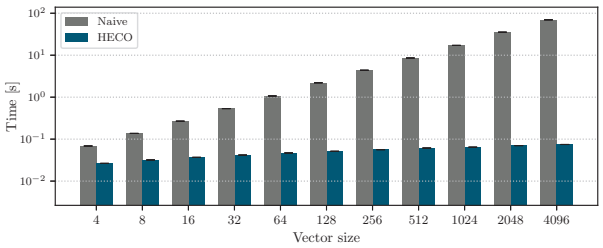
In contrast to existing monolithic compilers, HECO is highly modular and designed to be flexible and extensible. We decouple optimizations from front-end logic, allowing for a wide variety of domain-specific front-ends and the ability to easily replace back-ends to target different FHE libraries or hardware accelerators as they become available. The toolchain can easily be adapted to different needs, with certain optimizations enabled or disabled as required.

6 Evaluation

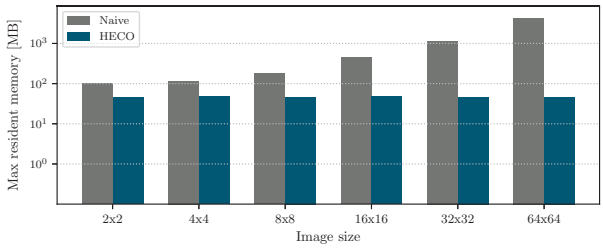
HECO is designed to compile high-level programs, written by non-experts in the standard imperative paradigm, into highly efficient batched FHE implementations that achieve the same performance as hand-crafted implementations by experts. HECO achieves its usability goals through a well-integrated Python front-end and by requiring developers to alter their code only minimally (annotating variables as secret). However, ease-of-use becomes moot when the performance of the generated code is not competitive. Therefore, we focus our evaluation on the performance of HECO and the code it generates, trying to answer whether or not automatic optimizations can bring naive code to the same performance level as expert implementations. In this section, we first show the effect of the batching optimizations on benchmark workloads designed to demonstrate different batching patterns, then compared against synthesized optimal batching patterns, and finally discuss a real-world application example.



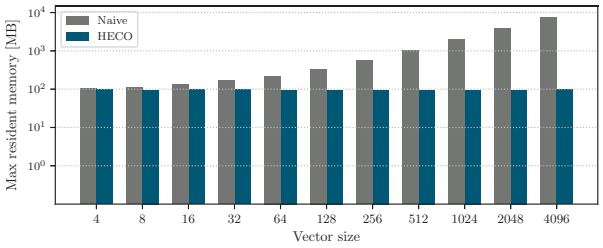
(a) Roberts Cross benchmark runtime (in seconds).



(b) Hamming Distance benchmark runtime (in seconds).



(c) Roberts Cross benchmark memory usage (in MB).



(d) Hamming Distance benchmark memory usage (in MB).

Figure 5: Log-log plot of the runtime and memory consumption of the roberts cross and hamming distance benchmarks for different vector sizes, comparing a naive non-batched solution with the batched solution generated by our system.

6.1 Benchmarks

We evaluate HECO in terms of the speedup reduction in memory overhead gained over non-optimized implementations and the compile time required.

Applications. We demonstrate the speedup achieved by our batching optimization on two applications that are representative of common batching opportunities. The *roberts cross operator* is an edge-detection feature used in image processing. It approximates the gradient of an image as the square root of the sum-of-squares of two different convolutions of the image, which compute the differences between diagonally adjacent pixels. As in all other kernel-based benchmarks, wrap-around padding is used, which aligns well with the cyclical rotation paradigm of FHE. In order to enable a practical FHE evaluation, the final square root is omitted, since it would be prohibitively expensive to evaluate under encryption. The *hamming distance*, meanwhile, computes the edit distance between two vectors, i.e., the number of positions at which they disagree. Here, we consider two binary vectors of the same length, a setting in which computing (non-)equality can be done efficiently using the arithmetic operations available in FHE. Specifically, this makes use of the fact that $\mathrm{NEQ}(a,b) = \mathrm{XOR}(a,b) = (a-b)^2$ for $a,b \in \{0,1\}$.

Baseline. Our baseline is a naive implementation of the application without taking advantage of batching, as one might expect FHE novices to implement. In this setting, vectors of secrets are directly translated to vectors of ciphertexts. While this approach introduces significant ciphertext expansion and increases the memory required, it is actually preferable in terms of run time over a solution that batches vector data into ciphertexts, but does not re-structure the program to be batching-friendly. This is because such a solution adds the overhead of rotations, masking, etc., to the base runtime of the non-batched solution.

Environment. All benchmarks are executed on AWS `m5n.xlarge` instances, which provide 4 cores and 16 GB of RAM. We used Microsoft SEAL [6] as the underlying FHE library, targeting its BFV [14, 15] scheme implementation. All experiments are run using the same parameters, which ensure at least 128-bit security. We report the run time of the computation itself, omitting client-side aspects such as key generation or encryption/decryption. All results are the average of 10 iterations, discarding top and bottom outliers.

Runtime & Memory Overhead. In Figure 5a we show the runtime of the Roberts Cross benchmark for varying instance sizes, comparing the non-batched baseline with the batched solution generated by HECO. While the run time of the naive version increases linearly with the image size, the batched solution maintains the same performance (until parameters must be increased to accommodate even larger images). Instead, the run time is more closely tied to the size of the kernel than to that of the image. This highlights the dramatic transformations achieved by HECO, fundamentally changing the structure of the program. As a result of these transformations, HECO achieves a speedup of 3454x over the non-batched baseline for 64x64 pixel images, demonstrating the extraordinary impact that effective use of batching can have on FHE applications: while the non-batched solution is borderline impractical at over two minutes, the batched solution takes only

a fraction of a second (0.04 s). The runtime of the generated code in this case does depend directly on the vector length. However, due to the fold-style optimization (cf. § 4.5), this dependence is only logarithmic.

In Figure 5d we can see that, for realistic problem sizes, the performance advantage of batching becomes significant, resulting in a speedup of 934x for 4096-element vectors. We also see that, while the runtime of the batched solution does increase with the vector length, this is nearly imperceptible when compared to the non-batched baseline. Finally, in Figure 5c/5b, we show the memory overhead of the non-batched baseline and the batched solution. While FHE introduces a non-negligible baseline overhead due to the large amount of key- and other context-data that must be maintained, the reduced number of ciphertexts in the batched solution has a clear impact on memory usage, increasingly so as the problem sizes increase.

Compile Time. HECO achieves these fundamental transformations efficiently, with compile times that are amenable to interactive development. This is in contrast to synthesis-based tools, which require more than 10 minutes to synthesize a batched solution for the Roberts Cross benchmark even for toy-sized instances and do not scale to the sizes we consider here at all [44].

6.2 Comparison with Synthesized Solutions

We compare the baseline and HECO to synthesized optimal batching patterns. Synthesis based approaches explore the space of all possible programs, constrained by a reference specification describing input-output behavior. While this tends to be computationally expensive, it has the potential to find optimal solutions featuring highly non-intuitive optimizations. We use a set of nine benchmark applications that represent a variety of common code patterns relevant to batching for our evaluation. These programs range from having no inter-element dependencies (e.g., Linear Polynomial), to simple accumulator patterns (e.g., Hamming Distance), and complex dependencies across a multitude of different vector elements (e.g., Roberts Cross). As a result, they provide a useful benchmark, especially for batching optimizations. These benchmarks were first proposed in [44], which introduces Porcupine [44], a synthesis-based compiler for batched FHE. In addition to a reference specification, it requires a developer-provided sketch of an initial possible batched approach. Our 'synthesis' solutions are based on pseudo-code made available in an extended version of the paper [44].

Synthesis based tools can require the significant search time to find solutions, limiting them to toy-sized workloads. For example, Porcupine requires over 10 minutes to synthesize a program with ten instructions and will fail to synthesize a solution at all for sufficiently complex programs. As a result, we consider the following problem sizes here: The synthesized dot-product code targets 8-element vectors, while those for Hamming Distance and L2 Distance were provided for 4-element vectors. For the other applications, we use vectors of length 4096, representing 64x64 pixel images

n	4	16	64	256	1024	4096
rc	0.06	0.07	0.08	0.12	0.30	1.28
hd	0.03	0.04	0.06	0.09	0.25	1.85

Table 1: Compile time (in seconds) of the Roberts Cross (rc) and Hamming Distance (hd) benchmarks for different problem sizes (n).

As we can see from Figure 6, HECO dramatically improves performance over the non-batched baseline approach. For example, for the Roberts Cross benchmark, our batched solution is over 3500 times faster than the non-batched solution, taking less than 0.04 seconds instead of over 2.35 minutes. More importantly, our results are nearly equivalent to the optimally batched solutions synthesized by Porcupine, especially when considering the stark contrast between the non-batched and the two batched solutions. In some cases (e.g., Box Blur), Porcupine has an advantage because it finds non-intuitive solutions beyond traditional batching patterns. Interestingly, for some applications (e.g. Hamming Distance) HECO actually outperforms Porcupine. This is because Porcupine provides an optimally batched solution but does not necessarily handle ciphertext management optimally, inserting unnecessary relinearization operations.

6.3 Real-World Application

The previous benchmarks demonstrated HECO's effectiveness and performance for different and common batching patterns. We now evaluate an application that more closely resembles the complexity that real-world settings exhibit. Specifically, we consider an application computing private statistics over two databases that might contain duplicate entries. We use this to demonstrate that HECO can produce efficient FHE code for non-trivial programs, while also highlighting that there is further room for optimizations exploiting application semantics.

Application. Privacy regulations frequently prohibit entities from combining sensitive datesets directly. Instead, they could employ *threshold* FHE, which extends FHE with multi-party key generation, to securely compute on the (encrypted) joint dataset. In this setting, neither party has sole access to the secret key, and they must collaborate to decrypt the results of approved queries. However, in practice their datasets might *overlap* (e.g., agencies at different levels of government collecting similar data), introducing duplicate items into the joint dataset. Due to the duplicates, analytics (e.g., counting

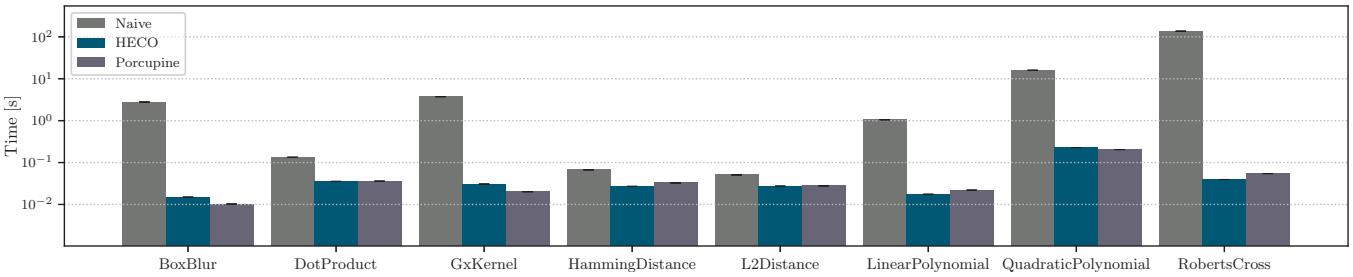


Figure 6: Runtime of example applications (in seconds), comparing a naive non-batched baseline, the solution generated by our system (HECO), and an optimally-batched solution synthesized by the Porcupine tool.

queries) will return incorrect results. Therefore, we must first de-duplicate the encrypted databases before executing the analytics. Since threshold FHE does not affect the server-side execution of the computation, HECO can be used directly to develop such an application. This first computes the *Private Set Union (PSU)* of two databases A,B indexed by unique IDs consistent across both (e.g., a national identifier like the SSN). For simplicity, we consider databases with one data column and a simple `SUM` aggregation. However, the presented approach trivially extends to larger databases and more complex statistics. Listing 5 shows the application expressed using the HECO Python frontend, for a database size of 128 elements and 8-bit identifiers. We split the identifiers into individual bits, allowing us to compute the equality function even while working with arithmetic circuits. The program begins by aggregating A's data and then proceeds to check each element of B's database for potential duplicates in A. In order to compute the equality function, we compute $\bigwedge_k a_k \oplus b_k$, using the fact that, for inputs $a, b \in \{0,1\}$, `xor` can be computed as $(a-b)^2$, and directly via multiplication, and `not` as $1-a$. If a duplicate is found, the element of B is multiplied with `unique == 0`, i.e., it does not contribute to the overall statistics.

```
def encryptedPSU(a_id: Tensor[128,8,Secret[int]],
                 a_data: Tensor[128,Secret[int]],
                 b_id: Tensor[128,8,Secret[int]],
                 b_data: Tensor[128,Secret[int]])
                 -> Secret[int]:
  sum: Secret[int] = 0
  for i in range(0, 128):
    sum = sum + a_data[i]
      for i in range(0, 128):
        unique: Secret[int] = 1
          for j in range(0, 128):
      # compute a_id[i] /= b_id[j]
    eq: sf64 = 1
    for k in range(0, 8):
       # a xor b == (a-b)^2
       x = (a_id[i][k] - b_id[j][k])**2
       nx = 1 - x  # not x
       eq = eq * nx # eq and nx
       neq = 1 - eq # not eq
       unique = unique * nequal

  sum = sum + unique * a_data[i]
  return sum
```

Listing 5: Computing statistics over duplicated data.

Performance & Discussion. We evaluated both a naive baseline and the HECO-optimized batched implementation using the same setup described earlier in this section. The naive approach has a run time of several minutes (11.3 *min*) and requires the sending of over 2000 ciphertexts between the server and the clients. The batched solution produced by our system requires not only significantly less data to be transmitted (only 4 ciphertexts), but also runs an order of magnitude faster (57.6 *s*), confirming the trend we observed when evaluating on smaller benchmarks. While the results achieved by HECO are more than practical already, a state-of-the art hand-written implementation designed for this task can improve this even further, requiring only 1.4 *s*. However, arriving at this solution requires a significant rethinking of the program and an application-specific batching pattern, which HECO intentionally does not consider to avoid a search space explosion. Note that synthesis based tools such as Porcupine also cannot capture these kinds of transformations.

Specifically, instead of batching the identifiers for each database into a single ciphertext, the expert solution instead creates one ciphertext per bit, using significantly oversized ciphertexts with $128^2 = 2^{14}$ slots. This enables the expert solution to batch every possible permutation of the identifier set into one ciphertext. By applying this to the encryption of set B, while simply encrypting 128 non-permuted repetitions of set A, the expensive (n^2) duplication check can be performed in parallel on all elements at the same time. Computing the `unique` flag then uses a rare application of the rotate-and-*multiply* pattern. As a trade-off, the equality computation is no longer batched, but since the number of bits in the identifiers is, by necessity, at most logarithmic in the number of database elements, this is a profitable trade-off.

In general, exploiting application-specific packing patterns can unlock additional performance gains and is a frequently combined with client-side processing in expert-designed FHE systems. However, the decision on which client-side processing (e.g., removing outliers, computing permutations, etc) is

sensible is not a well-defined problem that automated solutions can tackle. At the same time, the performance improvements demonstrated by HECO provide a first jump from the regime of prohibitive overheads to one of practical solutions. While further optimizations are likely frequently possible, they offer quickly diminishing returns. For example, scaling this application to real-world sizes (which, incidentally, is more complex with the expert approach) means that a naive solution might take days to compute while HECO's solution would complete in a few hours, which is a reasonable runtime for these kinds of secure statistics.

7 Related Work

In this section, we briefly discuss related work in the domain of FHE compilation (§ 7.1) followed by a discussion of differences to existing Multi-Party Computation (MPC) and Zero-Knowledge Proofs (ZKP) compilers (§ 7.2).

7.1 FHE Compilers

The complexity of implementing FHE operations efficiently led to the development of dedicated libraries [6, 48] early on. Today, a large number of libraries provide efficient implementations of state-of-the-art schemes. These libraries mostly provide comparatively low-level APIs that allow developers to extract the best possible performance but require significant expertise to utilize effectively. As a result, a first wave of FHE tools and compilers emerged that tried to improve the usability of FHE [31, 37, 49–51]. These mostly target a circuit-level abstraction and are focused on circuit optimizations [31].

For example, Microsoft's EVA [37] offers a user-friendly high-level interface and automatically inserts ciphertext maintenance operations into the circuit. EVA uses a custom circuit-based IR and requires developers to manually map their program to the FHE programming paradigm. In order to ease this process, recent versions [36] include a library of expert-implemented batched kernels for frequently used patterns, e.g., summing all elements in a vector. However, this still requires developers to manually transform an application to the batched paradigm.

A series of tools including Cingulata [33], E3 [51], SyFER-MLIR [52], and Google's Transpiler [53] attempt to translate arbitrary programs without the usual restrictions of FHE. They achieve this by translating their input programs into binary circuits, encrypting each input bit individually. However, programs translated in this way are virtually always too inefficient to be of practical use because they do not support the type of high-level transformations that HECO employs to achieve practical efficiency.

Domain-specific compilers [54–56], e.g., targeting encrypted Machine Learning applications, rely on a large set of hand-written expert-optimized kernels for common functionality (mostly linear algebra operations). Since these tools rely on pre-determined mappings rather than automatically identifying optimization opportunities, they do not transfer to other domains, such as the general-purpose setting HECO targets. Besides that, their lack of flexibility prevents their use when developers' needs are even slightly misaligned.

The Porcupine compiler [44] is closest to our work in that it also considers translating imperative programs to FHE's batching paradigm. However, their tool has a significantly different focus, using a heavy-weight synthesis approach that tries to identify optimal solutions that can outperform even state-of-the-art approaches used by experts. Since it explores a large state space in the search for an optimal solution, compile times tend to be long (up to many minutes) and programs can contain at most a handful of statements before the approach becomes infeasible. Additionally, Porcupine requires that developers provide a sketch of the structure of the batched program, making it less suitable for non-expert users.

Finally, we want to highlight that the MLIR framework is rapidly establishing itself as the gold standard for FHE tooling. Early attempts such as SyFER-MLIR [52] relied primarily on built-in optimizations, adding only a few binary-circuit-based FHE-specific rewrite rules. No evaluation is provided for SyFER-MLIR, but prior work studying similar simple rewrite rule-based tools [31] leads us to predict that it would produce only relatively minor speedups. More recently, however, a variety of concurrent work has successfully realized different aspects of the FHE ecosystem using MLIR. For example, Zama's Concrete-ML [57] internally uses an MLIR-based compiler to translate Machine Learning tasks expressed as Numpy programs to the TFHE scheme. HECATE [58], meanwhile, improves upon the rescale-allocation optimizations presented in the EVA compiler [37]. However, where the latter uses a custom Python implementation that does not provide for interoperability with other tools, HECATE builds upon common MLIR abstractions. Thanks to the modular nature of MLIR, these tools could easily be integrated into HECO's end-to-end toolchain.

7.2 MPC & ZKP Compilers

Compilers for both Multi-Party Computation (MPC) and Zero-Knowledge Proof (ZKP) systems face similar challenges to FHE compilation, such as the need for data-independent computations and a general tendency towards trying to achieve small and low-depth circuits. However, in practice we find these similarities are too superficial to allow techniques from one domain to be lifted to another. Due to the heavy reliance on selectively revealing (potentially blinded) data during an MPC computation – a feature that has no direct correspondence in FHE – many of the optimization approaches are unlikely to transfer. State-of-the art MPC compilers [59, 60] also frequently make heavy use of hybrid approaches, i.e., switching between different MPC settings. While there has been significant work on scheme switching for FHE [25, 26],

practical applications remain rare and few libraries currently support these techniques. As these approaches start to mature, investigating to what extent scheme-switching optimizations from the domain of MPC can transfer to FHE will present an interesting avenue for future work.

Zero-Knowledge Proof Compilers also face the challenge of mapping complex operations to arithmetic circuits with limited expressiveness. However, their setting fundamentally differs from that of FHE, as the prover generally has access to all data in the computation in the clear. This allows ZKP computations to heavily rely on witness-based computation, which allows compilers to shift virtually all non-arithmetic operations outside the core ZKP computation. Additionally, ZKP compilers mostly use intermediate representations based on Rank-1 Constraint Systems (R1CS) or other constraint specification systems that are not suitable for expressing FHE computations. Finally, we note that compiler frameworks trying to accommodate MPC, ZKP and potentially also FHE are emerging [61]. However, their reliance on a circuit-like IR makes them unsuitable for the high-level transformations we use in our work.

8 Discussion

As FHE has emerged into practicality, it has drawn the interest of a significantly wider audience bringing new perspectives, requirements and backgrounds to the area. While traditionally, FHE applications were mostly developed by the same experts that designed, optimized and implemented the underlying cryptographic schemes, this will soon no longer be true beyond the world of cutting-edge academic research. In recent years, a variety of tools has emerged in an attempt to address the needs of future non-expert developers. While some domain specific tools have proven to be very effective [31, 56, 62], most general purpose tools have fallen short of delivering on the promise of usable FHE. While they simplify the development process, they generally produce naive implementations which provide little real-world benefit due to their significant overhead compared to more optimal implementations. However, as performance is key for practical deployment, we believe that usability without sufficient performance is mostly meaningless.

HECO aims to bridge the gap between usability and performance for general-purpose workloads. It offers non-expert developers the ability to express applications in a familiar high-level paradigm *without* paying the extreme performance penalty this would usually incur. Beyond optimizing this high-level transformation, HECO proposes a new end-to-end architecture for FHE compilers based on the distinct stages of FHE optimization we identify. HECO's modular architecture is designed to allow it to interoperate with other toolchains and easily integrate future optimization techniques. While HECO represents an important step in FHE usability, many challenges remain to be addressed. For example, HECO considers RLWE-based schemes offering SIMD operations, but recent developments in LWE-based fast-bootstrapping schemes makes them an attractive alternative. Today, these worlds remain mostly separate and use significantly different paradigms. Future work needs to consider how to unify these, especially in the context of scheme-switching, i.e., the ability to move between schemes inside a single application. Finally, upcoming dedicated FHE hardware accelerators promise to deliver significant performance improvements but require sophisticated scheduling to unlock their potential. While existing work on accelerators already incorporates automated scheduling, there are likely significant further optimization opportunities in considering compilation for these systems from an end-to-end perspective. More generally, we believe that there is significant potential for interdisciplinary research that combines techniques from compiler and programming language research with insights from cryptography.

Acknowledgments

We thank Gyorgy Rethy and Nicolas Küchler for their invaluable help in evaluating HECO. We also thank our anonymous reviewers, Tobias Gross, Michael Steiner, and the PPS Lab team for their insightful input and feedback. We would also like to acknowledge our sponsors for their generous support, including Meta, Google, SNSF through an Ambizione Grant No. 186050, and the Semiconductor Research Corporation.

References

[1] S. Kannepalli, K. Laine, and R. C. Moreno, "Password monitor: Safeguarding passwords in microsoft edge." https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/, 21 Jan. 2021. Accessed: 2021-7-5.

[2] M. Kim, A. O. Harmanci, J.-P. Bossuat, S. Carpov, J. H. Cheon, I. Chillotti, W. Cho, D. Froelicher, N. Gama, M. Georgieva, S. Hong, J.-P. Hubaux, D. Kim, K. Lauter, Y. Ma, L. Ohno-Machado, H. Sofia, Y. Son, Y. Song, J. Troncoso-Pastoriza, and X. Jiang, "Ultrafast homomorphic encryption models enable secure outsourcing of genotype imputation," *Cell systems*, vol. 12, pp. 1108–1120.e4, 17 Nov. 2021.

[3] S. Carpov, N. Gama, M. Georgieva, and J. R. Troncoso-Pastoriza, "Privacy-preserving semi-parallel logistic regression training with fully homomorphic encryption," *BMC medical genomics*, vol. 13, p. 88, 21 July 2020.

[4] DARPA, "Data protection in virtual environments (DPRIVE)." https://sam.gov/opp/16c71dadbe814127b475ce309929374b/view, 27 Feb. 2020.

[5] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, (New York, NY, USA), pp. 238–252, Association for Computing Machinery, 18 Oct. 2021.

[6] "Microsoft SEAL (release 3.5)," Apr. 2020.

[7] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: A compiler infrastructure for the end of moore's law," 25 Feb. 2020, arXiv:cs.PL/2002.11054.

[8] P. Paillier, "Public-Key cryptosystems based on composite degree residuosity classes," in *Advances in Cryptology — EUROCRYPT '99*, EUROCRYPT, (Prague, Czech Republic), pp. 223–238, Springer, Berlin, Heidelberg, May 1999.

[9] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, Feb. 1978.

[10] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," in *Proceedings of the 41st Annual ACM Symposium on Symposium on Theory of Computing - STOC '09*, (Bethesda, MD, USA), p. 169, ACM Press, 2009.

[11] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data," in *ISCA*, pp. 173–187, 2022.

[12] R. Cammarota, "Intel HERACLES: Homomorphic encryption revolutionary accelerator with correctness for learning-oriented End-to-End solutions," in *Proceedings of the 2022 on Cloud Computing Security Workshop*, CCSW'22, (New York, NY, USA), p. 3, Association for Computing Machinery, 7 Nov. 2022.

[13] R. Geelen, M. Van Beirendonck, H. V. L. Pereira, B. Huffman, T. McAuley, B. Selfridge, D. Wagner, G. Dimou, I. Verbauwhede, F. Vercauteren, and D. W. Archer, "BASALISC: Flexible asynchronous hardware accelerator for fully homomorphic encryption," 27 May 2022, arXiv:cs.CR/2205.14017.

[14] Z. Brakerski, "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP," in *Advances in Cryptology – CRYPTO 2012*, vol. 7417, pp. 868–886, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[15] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.

[16] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, (New York, NY, USA), p. 309–325, Association for Computing Machinery, 2012.

[17] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *Advances in Cryptology – ASIACRYPT 2017*, vol. 10624, pp. 409–437, Cham: Springer International Publishing, 2017.

[18] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, 1 Jan. 2015.

[19] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan, "Homomorphic encryption security standard," tech. rep., HomomorphicEncryption.org, Toronto, Canada, Nov. 2018.

[20] L. Brenna, I. S. Singh, H. D. Johansen, and D. Johansen, "TFHE-rs: A library for safe and secure remote computing using fully homomorphic encryption and trusted execution environments," *Array*, p. 100118, 28 Dec. 2021.

[21] F. Bourse, R. Del Pino, M. Minelli, and H. Wee, "FHE circuit privacy almost for free," in *Advances in Cryptology – CRYPTO 2016*, pp. 62–89, Springer Berlin Heidelberg, 2016.

[22] L. Ducas and D. Stehlé, "Sanitization of FHE ciphertexts," in *Advances in Cryptology – EUROCRYPT 2016*, Lecture notes in computer science, pp. 294–310, Berlin, Heidelberg: Springer Berlin Heidelberg, 2016.

[23] B. Li and D. Micciancio, "On the security of homomorphic encryption on approximate numbers," in *Advances in Cryptology – EUROCRYPT 2021*, (Cham), pp. 648–677, Springer International Publishing, 2021.

[24] B. Li, D. Micciancio, M. Schultz, and J. Sorrell, "Securing approximate homomorphic encryption using differential privacy," *Cryptology ePrint Archive*, 2022.

[25] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, "Chimera: Combining ring-lwe-based fully homomorphic encryption schemes," *Journal of Mathematical Cryptology*, vol. 14, no. 1, pp. 316–338, 2020.

[26] W. jie Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu, "PEGASUS: Bridging polynomial and non-polynomial evaluations in homomorphic encryption," in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2021.

[27] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks." Cryptology ePrint Archive, Report 2021/091, 2021. https://ia.cr/2021/091.

[28] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, (New York, NY, USA), pp. 238–252, Association for Computing Machinery, 18 Oct. 2021.

[29] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data," in *ISCA*, pp. 173–187, 2022.

[30] R. Geelen, M. Van Beirendonck, H. V. L. Pereira, B. Huffman, T. McAuley, B. Selfridge, D. Wagner, G. Dimou, I. Verbauwhede, F. Vercauteren, and D. W. Archer, "BASALISC: Flexible asynchronous hardware accelerator for fully homomorphic encryption," 27 May 2022, arXiv:cs.CR/2205.14017.

[31] A. Viand, P. Jattke, and A. Hithnawi, "SoK: Fully homomorphic encryption compilers," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 1166–1182, 2021.

[32] D. W. Archer, J. M. Calderón Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan, "RAMPARTS: A Programmer-Friendly system for building homomorphic encryption applications," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography - WAHC'19*, (New York, New York, USA), pp. 57–68, ACM Press, 2019.

[33] 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*, SCC '15, (New York, NY, USA), pp. 13–19, ACM, 2015.

[34] S. Carpov, P. Aubry, and R. Sirdey, "A multi-start heuristic for multiplicative depth minimization of boolean circuits," in *Combinatorial Algorithms*, pp. 275–286, Springer International Publishing, 2018.

[35] P. Aubry, S. Carpov, and R. Sirdey, "Faster homomorphic encryption is not enough: Improved heuristic for multiplicative depth minimization of boolean circuits," in *Topics in Cryptology – CT-RSA 2020*, pp. 345–363, Springer International Publishing, 2020.

[36] S. Chowdhary, W. Dai, K. Laine, and O. Saarikivi, "EVA improved: Compiler and extension library for CKKS," in *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '21, (New York, NY, USA), pp. 43–55, Association for Computing Machinery, 15 Nov. 2021.

[37] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "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*, 27 Dec. 2019.

[38] Y. Lee, S. Heo, S. Cheon, S. Jeong, C. Kim, E. Kim, D. Lee, and H. Kim, "HECATE: Performance-Aware scale optimization for homomorphic encryption compiler," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 193–204, Apr. 2022.

[39] A. Costache, K. Laine, and R. Player, "Evaluating the effectiveness of heuristic worst-case noise analysis in FHE." Cryptology ePrint Archive, Report 2019/493, 2019.

[40] I. Iliashenko, "Optimisations of fully homomorphic encryption," 27 May 2019. PhD. Thesis, KU Leuven.

[41] A. Costache, B. R. Curtis, E. Hales, S. Murphy, T. Ogilvie, and R. Player, "On the precision loss in approximate homomorphic encryption." https://eprint.iacr.org/2022/162.pdf. Accessed: 2022-2-26.

[42] A. Costache, K. Laine, and R. Player, "Evaluating the effectiveness of heuristic worst-case noise analysis in fhe," in *Computer Security – ESORICS 2020*, pp. 546–565, Springer, Sept. 2020.

[43] F. Boemer, S. Kim, G. Seifu, F. D. M. de Souza, and V. Gopal, "Intel HEXL: Accelerating homomorphic encryption with intel AVX512-IFMA52," 30 Mar. 2021, arXiv:cs.CR/2103.16400.

[44] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, "Porcupine: A synthesizing compiler for vectorized homomorphic encryption," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, (New York, NY, USA), p. 375–389, Association for Computing Machinery, 2021.

[45] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," *SIGPLAN Not.*, vol. 35, pp. 145–156, 1 May 2000.

[46] C. Mendis and S. Amarasinghe, "goSLP: globally optimized superword level parallelism framework," *Proc. ACM Program. Lang.*, vol. 2, pp. 1–28, 24 Oct. 2018.

[47] Y. Chen, C. Mendis, M. Carbin, and S. Amarasinghe, "VeGen: a vectorizer generator for SIMD and beyond," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, (New York, NY, USA), pp. 902–914, Association for Computing Machinery, 19 Apr. 2021.

[48] S. Halevi and V. Shoup, "Design and Implementation of a Homomorphic-Encryption Library," *IBM Research (Manuscript)*, vol. 6, pp. 12–15, 2013.

[49] A. Viand and H. Shafagh, "Marble: Making fully homomorphic encryption accessible to all," in *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pp. 49–60, ACM, Oct. 2018.

[50] 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*, SCC '15, (New York, NY, USA), pp. 13–19, ACM, 2015.

[51] E. Chielle, N. G. Tsoutsos, O. Mazonka, and M. Maniatakos, "Encrypt-Everything-Everywhere: ISA extensions for private computation," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2020.

[52] S. Govindarajan and W. S. Moses, "SyFER-MLIR: Integrating Fully Homomorphic Encryption Into the MLIR Compiler Framework," 2020.

[53] S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. J. 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," *CoRR*, vol. abs/2106.07893, 2021, :/2106.07893.

[54] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "CHET: an optimizing compiler for fully-homomorphic neural-network inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, (New York, NY, USA), pp. 142–156, ACM, 8 June 2019.

[55] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "nGraph-HE2: A High-Throughput framework for neural network inference on encrypted data," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'19, (New York, NY, USA), pp. 45–56, Association for Computing Machinery, Nov. 2019.

[56] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, CF '19, (New York, NY, USA), pp. 3–13, ACM, 2019.

[57] Zama, "concrete-numpy: Concrete numpy is an open-source library which simplifies the use of fully homomorphic encryption."

[58] Y. Lee, S. Heo, S. Cheon, S. Jeong, C. Kim, E. Kim, D. Lee, and H. Kim, "HECATE: Performance-Aware scale optimization for homomorphic encryption compiler," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 193–204, Apr. 2022.

[59] N. Büscher and S. Katzenbeisser, *Compilation for Secure Multi-party Computation*. SpringerBriefs in Computer Science, Springer International Publishing, 2017.

[60] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "SoK: General purpose compilers for secure Multi-Party computation," in *IEEE Symposium on Security and Privacy (SP)*, (Los Alamitos, CA, USA), pp. 479–496, IEEE Computer Society, 2019.

[61] A. Ozdemir, F. Brown, and R. S. Wahby, "CirC: Compiler infrastructure for proof systems, software verification, and more," in *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 2248–2266, May 2022.

[62] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks." Cryptology ePrint Archive, Report 2021/091, 2021.