HECO: Automatic Code Optimizations for Efficient Fully Homomorphic Encryption

Alexander Viand
Department of Computer Science
ETH Zurich, Switzerland

Patrick Jattke
Department of Electrical Engineering
ETH Zurich, Switzerland

Miro Haller
Department of Computer Science
ETH Zurich, Switzerland

Anwar Hithnawi
Department of Computer Science
ETH Zurich, Switzerland

Abstract

In the last decade, Fully Homomorphic Encryption (FHE) has evolved from an unrealized cryptographic concept with the premise of enabling delegation of computation, without the need to give access to the data, to a practical technology that today has the potential to transform the digital world to a more secure and private one. More recently, FHE has undergone several breakthroughs and advancements that led to a leap in performance improvements, making FHE performance no longer a major barrier to its 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 outside the cryptographic community. In addition, existing FHE compilers tend to consider optimizations in isolation, focusing on a specific target scenario. They are usually implemented as monolithic tools, defining a single compilation pipeline including optimizations relevant to their application scenario. Given the rapidly evolving field, and the potential for combining and adapting optimizations across schemes and settings, this narrow approach is unnecessarily limiting. In this paper, we present HECO, a new end-to-end design for FHE compiler 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 immediate cryptographic challenges that existing tools have focused on.

1 Introduction

Privacy and security are gaining tremendous importance across all organizations, as public perception of these issues 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 confidentiality while in transit, rest, and in-use. FHE is one key secure computation technology that enables systems to preserve the confidentiality of data end-to-end, including while in use. Hence, allowing outsourcing of computation without having to grant access to data. In the last decade, advances in FHE schemes have propelled FHE from a primarily theoretical breakthrough to a practical solution for a wide range of applications [7, 26, 27] With dedicated hardware accelerators for FHE on the horizon promising further speedup [14], FHE will soon be competitive for a wider set of applications. In recent years, we have seen FHE emerge in real-world products, e.g., Microsoft's Edge browser uses FHE to realize its privacy-preserving password monitor.

Though promising in its potential, developing FHE applications unfortunately remains a complex and tedious process. Its use in practice and at scale is today primarily hindered by its complexity rather than its performance. Writing efficient FHE programs that deliver on its potential performance is currently a complicated and tedious task that even experts struggle with. These complexities are attributed to the a unique programming paradigm FHE imposes (In Section 3 we discuss this paradigm and its implication in more detail). Also, performance characteristics of FHE are non-intuitive, highly contextual, and anticipating them requires a deep understanding of the underlying schemes. Performance differences between unoptimized code and code that has been adapted to the FHE paradigm can be overwhelmingly large, with differences by more than one Order of Magnitude (OoM) being common. Even with improvements in the performance of the underlying schemes appear, the need to generate optimized FHE code continue to have a key influence on applications performance. Therefore, optimizing applications and rewriting them to match FHE's programming paradigm is essential for practical applications of FHE. Today, this remains an unsolved issue that poses a major barrier to wider adoption of FHE.

Recently, tools have started to emerge that aim to address some of the complexities FHE impose. However, none of these tools solve the fundamental issues arising from the

unique programming paradigm of FHE [35].

The unique programming paradigm of FHE poses serious barriers to entry, and restricts the development of efficient and secure FHE applications to a small pool of experts familiar with this paradigm and the 'folklore' of the various techniques used to fully exploit the potential of modern FHE schemes. Enabling wider adoption of FHE will require tools that abstract away these complexities from developers by automatically translating their high-level code into efficient and secure FHE implementations. This automatic compilation of efficient FHE kernels is currently lacking in existing FHE compilers.

The core of this work, and hence its *first contribution*, is expanding the FHE compilation chain with support for automatic transformation of high-level programs to FHE's unique programming paradigm. Realizing this form of code transformation necessitate that we rethink the existing flow of FHE compilers. Existing compilers either naively translate high-level programs into low-level circuits of native FHE operation, or require developers to already provide paradigm-compliant programs. However, the mapping between high-level programs and circuit-like FHE programs is a significant opportunity for optimizations. As a result, the vast majority of expert optimizations happen at this level, including algorithmic adaptation such as batching. Trying to identify opportunities to apply such optimizations in a naively converted circuit is nearly impossible. Instead, we need to apply these transformations while we still have access to the semantics of a high-level program. However, this requires extending the scope of FHE compilers from circuit generators and optimizers to true compilers applying optimizations to high-level *programs* rather than circuit.

Core to this transformation logic we introduce and incorporate a new FHE *batching* code transformation that exploits the Single Instruction, Multiple Data (SIMD) parallelism present in most modern FHE schemes (Section 5)).

In addition to introducing automatic code compilation logic, we present a new end-to-end design of FHE compiler that that support easy adaptation of the compilation process to different application scenarios and different FHE schemes optimizations, including potential future ones. Existing FHE compilers tend to consider optimizations in isolation, focusing on a specific target scenario. They are usually implemented as monolithic tools, defining a single compilation pipeline including optimizations relevant to their application scenario. This results in implementations that are tied closely to a specific scheme, or even a specific software implementation of a scheme. While some optimization techniques are highly scheme or application specific, many others are nearly universally applicable. However, the approach taken by existing compilers prevents the re-use of optimization techniques, since they are tightly integrated into these monolithic tools. Instead, we bring a modular design that decouples optimizations from front-end logic, allowing for a wide variety of domain-specific frontends and the ability to easily replace the backend to target different FHE libraries or hardware accelerators as they become available. This way, the toolchain can easily adapted to different needs, with certain optimizations enabled or disabled as required, and new optimizations easily added. We have built a prototype of HECO[1] and evaluated it on common FHE application benchmarks. Our evaluation results show that HECO can provide orders of magnitude in performance gains when translating code that has been expressed in the traditional imperative paradigm.

2 Background

In this section, we briefly introduce the notion of Fully Homomorphic Encryption (FHE) and its threat model.

2.1 Fully Homomorphic Encryption

In a *homomorphic* encryption scheme there exists a homomorphism between operations on the plaintext and operations on the ciphertext. While *additively* or *multiplicative* homomorphic encryption schemes such as the Paillier encryption scheme [31] or textbook RSA [33] have been known for many decades, *fully* homomorphic schemes that support an arbitrary combination of both operations remained unrealized until 2009 when Craig Gentry presented the first feasible FHE scheme [19]. When used over a binary plaintext space ($\mathbb{Z}_2$), multiplication and addition emulate `AND`- and `XOR`-gates and can emulate arbitrary computations [34].

2.2 FHE Schemes

An FHE scheme, is an encryption scheme that allows unlimited number of arbitrary evaluation over the ciphertexts. Modern FHE schemes broadly follow a general approach laid out by Gentry's initial scheme and are mostly based on the hardness of the Learning with Errors (LWE) problem [32] or the related Ring-LWE (RLWE) problem [30].

In RLWE, plaintexts and ciphertexts are polynomials of degree n (usually $n > 2^{12}$). While this can be used to encrypt single integers, there are more efficient encoding approaches, known as *batching* that utilize the entire polynomial. Security in the LWE setting derives from the presence of small *noise* in the ciphertext. This is initially small enough to be rounded away, but grows during homomorphic operations. If left unmanaged, the nose will eventually corrupt the ciphertext, leading to failed decryptions. While this effect is negligible during additions, multiplying two ciphertexts introduces significantly more noise. This limits computations to a (parameter-dependent) number of consecutive multiplications (multiplicative *depth*) before decryption fails. This limitation

[1] HECO's code is available at: https://github.com/pps-lab/nonymized-for-review

can be circumvented using *bootstrapping*, which resets the noise level of a ciphertext to a fixed lower level by homomorphically evaluating the decryption circuit with an encrypted secret key as input. However, the decryption circuit needs to be sufficiently low-depth to allow at least one additional multiplication before needing to bootstrap again. Since bootstrapping is computationally very expensive, many practical applications instead choose the parameters large enough to perform the desired computation without bootstrapping being required, which is known as *levelled* FHE.

Important Schemes In addition to schemes like Brakerski/Fan-Vercauteren (BFV) [3, 17], Brakerski-Gentry-Vaikuntanathan (BGV) [4], and Cheon-Kim-Kim-Song (CKKS) [8] that follow similar batching-focussed constructions, there are also schemes like Fast Fully Homomorphic Encryption Library over the Torus (TFHE) [10, 11] that derive from the Gentry-Sahai-Waters (GSW) scheme [20]. These schemes focus on fast bootstrapping, but in turn give up the ability to batch, making them less suitable for scenarios with large amounts of data. While initially limited to binary settings, recent follow-up work [12] extends these schemes to arithmetic circuits and introduces the notion of *programmable* bootstrapping to approximate non-polynomial functions. Recently, there has also been work that explores homomorphic conversions between different schemes [2, 25] and using these, it might be possible to combine the advantages of both approaches.

2.3 Threat Model

Our compiler generally inherits the threat models of the underlying schemes that a developer might target. Specifically, we point out that FHE does not by default provide *circuit privacy*, i.e., a client might be able to learn the circuit the server applied Additionally, it is worth noting that FHE does not provide *integrity* by default, i.e., a server might perform a different calculation than the client requested or even no calculation at all. Finally, there are some subtle issues that can appear when using *approximate* homomorphic encryption (e.g., CKKS). These arise when a third-party gains access to a large number of high-precision decryptions where it knows the input and the comwputation that was performed to arrive at this point [29]. While this situation is unlikely to appear in many scenarios since releasing the decrypted result to an untrusted party somewhat undermines the point of using FHE in the first place, it should nevertheless be mentioned.

3 FHE Programming Paradigm

Developing applications for FHE is fundamentally different from writing plaintext program, as FHE impose a unique programming paradigm. Parts of this paradigm shift derive directly from the definition of FHE and its security guarantees, while others are due to common characteristics nearly all modern FHE schemes share. Developing applications that meets the performance potential of FHE requires understanding the different cost model FHE impose but also the unique optimization opportunities the underlying schemes provide. As result, developing secure and efficient FHE solutions today is notoriously hard and remains limited to experts.

In this section, we analyze and discuss the programming challenges FHE imposes, highlighting their consequences on the developers experience and illustrate the techniques experts use to achieve high performance in this unique computational and programing paradigm. Where relevant, we discuss how these complexities can be addressed using automatic compilation and optimization, which is the focus of this paper.

3.1 Data Independent Computation

The security properties of FHE guarantee that user inputs and intermediate values remain secret, even from the server performing the computation. This implies that we cannot branch based on secret inputs as revealing even the single bit required to branch would leak information about the encrypted data. Therefore, all FHE computations must be *data-independent*. Emulating, e.g., if/else branches is possible by calculating the result for both branches and performing a multiplexing selection afterward. However, this requires evaluating both branches. *Consequences:* Nearly all modern programming relies heavily on branching, either explicitly (e.g., If/Else statements and dynamic-length loops in iterative programming) or implicitly (e.g., recursion base cases in functional programming). As a result, developers need to switch their approach to writing applications quite significantly when working with FHE. Without data-dependent branching, the runtime of a program is fixed to (at least) the worst-case runtime of the program. In fact, since we have to evaluate *both* possible paths of each branching operation, the runtime of a data-independent version is usually even worse than the worst-case runtime of the original program. To circumvent this overhead, well-written FHE applications tend to eliminate branching or, where this is impossible, design algorithms directly in terms of conditional assignments, expressed as `x = c*a + (1-c)*b` for $\text{Dec}(\texttt{c}) \in \{0,1\}$.

What support existing FHE tools provide to instrument this shift of programming paradigm? Few existing tools support branching, with most requiring the input program to be data-independent already. Those that do support data-dependent branching tend to naively translate it directly to circuits. Instead of asking users to shift their paradigm when programming, we must try to rewrite branching-based programs to be as efficient as possible. In order to reduce the impact of branching on performance, we must minimize the duplication of work that occurs. Given a circuit representation of a program, it is very difficult to reconstruct branching behavior and

try to optimize it. Instead, we need to apply static analysis and optimizations on the original program in order to reduce the number of variables that are affected by branching. In HECO, we support branching in the input program and automatically translate it so that developers do not have to to change their paradigm.

3.2 Datatypes & Operations

In addition to restricting data-flow, FHE schemes also offer a very limited selection of data types and operations. The basic operations of FHE are additions and multiplications. While, over $\mathbb{Z}_2$, this can emulate arbitrary computation, the best performance is usually achieved when using non-binary plaintext spaces. Apart from a few schemes designed specifically for binary ($\mathbb{Z}_2$) or to represent approximations of fractional numbers ($\mathbb{R}, \mathbb{C}$), most schemes and implementations focus on integers ($\mathbb{Z}_t$ for $t \gg 2$), restricting computation to polynomial functions. One also need to consider the noise growth that homomorphic operations introduce to the ciphertext. Schemes offer *ciphertext maintenance* operations which do not modify the encrypted value, but either lower the noise (i.e., bootstrapping) or reduce future noise growth. Using these effectively requires a thorough understanding of the underlying cryptographic primitives.

Consequences. Traditional encryption schemes transparently translate messages into and out of the scheme's plaintext space (e.g., curve points). Beyond serializing messages into binary strings, developers generally do not need to worry about encodings or consider the plaintext space and its semantics. In FHE, however, the plaintext space also determines the semantics of the computation. As a result, choosing an appropriate encoding for one's messages is an important step of the FHE development process. Choosing poorly could lead to programs that are hundreds of time slower or even produce incorrect results. While binary bit-wise encryption can be used to represent more complex operations or non-numeric data, this is generally highly inefficient. As a result, the majority of practical FHE applications use arithmetic circuits over integers, where we are limited to polynomial functions. While it is possible to approximate many functions using polynomials, standard approximation approaches are frequently prohibitively computationally expensive in the context of FHE. Instead, efficient FHE implementations need to reformulate their computation, e.g., using different algorithms that are either inherently polynomial or well-suited to low-degree polynomial approximations. Tool support for such re-writing does not currently exist, and might be impossible in the general case, as these transformation require a deep understanding of the application domain and requirements.

3.3 SIMD Parallelism

In many modern FHE schemes, the plaintext space is a ring of polynomials with many (usually 2^{13}–2^{16}) coefficients. Instead of encrypting a single message into these polynomials, these schemes introduced support for Single Instruction, Multiple Data (SIMD)-style *batching*, which allows one to encrypt many messages into the same ciphertext and operate on them at the same time. Directly using the coefficients of the plaintext/ciphertext polynomials as the vector slots would lead to undesirable interactions between different slots during homomorphic multiplications. Instead, the Chinese Remainder Theorem [24] is used to reinterpreted the ring as a product of many rings, making the slots independent from each other. However, SIMD parallelism is limited without the ability to let data from different slots interact with each other. In FHE, this is realized through automorphisms which homomorphically rotate values in a ciphertext between slots in a cyclical fashion [22].

Consequences When possible, batching is by far the most important optimization technique, since forgoing it means that using only $1/n$ (e.g., $1/16k$) of the computational capacity. While it os trivial to use SIMD batching to increase throughput by batching many inputs together, this is frequently not feasible in practice since many applications either do not feature enough inputs to fill a large ciphertext vector or are primarily latency limited. The challenge, therefore, is to use SIMD to accelerate the computation on a single (or a small number of) input(s). Since there are no no native scatter/-gather operations, once a ciphertext has been encrypted, we are limited to cyclical rotations and SIMD operations acting over all slots. The challenge of this paradigm arises from the complexity of efficiently computing functions where values from many different slots need to interact with each other. Note that while there are more complex encoding approaches, beyond the SIMD technique that we consider here, these are very application specific and fairly uncommon.

Tooling Existing tools generally do not offer developers support in mapping programs to this complex model. However, the difference between naive non-batched and expertly batched programs is dramatic [35]. This mapping, however, requires understanding FHE cost model very well and a lot of intuition, experience and exposure to folklore about techniques and algorithms in this paradigm.

4 HECO Architecture

In this section, we provide a system overview of HECO and then present its components.

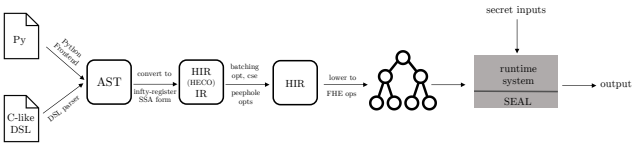

Figure 1: Overview of HECO's end-to-end approach.

4.1 System Overview

In HECO, developers express their programs using high-level languages (Python or a C-like Domain-Specific Language (DSL)). In contrast to other tools, we extract the entire Abstract Syntax Tree (AST) of the input program, rather than a circuit-like trace. This allows us to apply high-level transformations, like mapping imperative programs to the unique programming paradigm of FHE. Rather than applying these transformations on the AST, as is done in traditional compilers (e.g., gcc, clang), we translate the AST into a high-level Intermediate Representation (IR), following modern approaches to compiler design. This IR represents control-flow using high-level constructs, rather than the branch-and-label approach common in lower-level IRs, allowing us to easily identify and transform, e.g, loops and If/Else statements. Working over a high-level IR also allows HECO to introduce a clean separation of concerns and be frontend-agnostic: any frontend that can translate its input programs into the high-level IR can be utilized without requiring any modifications to HECO.

Optimizations in HECO are implemented as self-contained passes over this high-level IR, which can be composed dynamically into different compilation pipelines. Passes include our SIMD batching optimizations, but also standard optimizations such as Common Subexpression Elimination (CSE) and dead code elimination. These optimization passes rewrite the program, but stay in the same high-level IR. In contrast, conversion passes progressively *lower* the program to a more concrete, lower-level representation. Here, HECO uses an HE IR that represents the different native FHE operations offered by FHE schemes, including the various ciphertext maintenance operations. This lower-level HE IR is still a register-based Static-Single Assignment (SSA) representation of a *program*, however, a program in this IR is conceptually equivalent to a circuit. Therefore we refer to optimizations that act on the HE IR level as *circuit* optimizations. Existing work, which so far has focused exclusively on this level, has proposed a variety of technique at this level and we have implemented some existing techniques (e.g., ciphertext maintenance operation insertion) in HECO. The modularity of our system, allows HECO to be easily extendable with further optimizations, including potential future ones.

Finally, HECO translates the generated, optimized, HE IR into a concrete implementation for a specific backend. HECO is as back-end agnostic as it is fron-end agnostic: Since the HR IR corresponds directly to native FHE operations, it is trivial to generate code targeting a specific FHE implementation. Currently, HECO uses Microsoft's Simple Encrypted Arithmetic Library (SEAL) as its primary backend. In addition to producing fully compiled and linked binaries, HECO can also emit `C++` files. This allows advanced users to analyze and potentially modify the generated code, and allows users to integrate HECO into their existing development toolchain. Beyond targeting different libraries directly, the level of abstraction used in the HE IR also corresponds to the interface upcoming hardware accelerators will use [18], making them an obvious future backend to target. Figure 1 shows an overview over HECO's end to end approach. In the remainder of this section we provide an overview of language abstractions, optimizations, and runtime backends before describing our batching optimization algorithm that underpins the the automatic code translation in HECO (Section 5).

4.2 Frontend & High-Level IR

HECO allows developers to express their programs in the same imperative paradigm they are used to, using high-level languages. In addition to a custom C-like Domain-Specific Language (DSL), we provide a Python-based frontend which allows developers to work with FHE in a familiar environment. We use Python type hints to annotate which inputs are *secret*, and provide overloaded operators. Though our design targets primarily non-expert developers, we expect that expert developers will also benefit greatly from the reduced complexity and shorter time-to-solution of FHE tools. Therefore, HECO's frontend also exposes more advanced features like rotations and ciphertext maintenance operations through built-in functions that seamlessly integrate with the rest of the frontend language.

Extracting ASTs. Since HECO operates over programs, rather than circuits, we need to preserve this control-flow and extract the entire program AST. While this is trivial for a DSL, the user experience of niche DSLs is frequently poor, since there is limited syntax highlighting support and no pre-existing ecosystem of editor/IDE integration. A Python-based frontend solves these issues, but introduces new challenges. Existing tools with Python interfaces use a *dummy* object that records operations applied to it, building up a large expression tree for the output of the computation [15]. However, in this process, information about the control-flow and structure of the program is lost and we retain only the circuit representation of the input program. Instead, HECO's front-end uses Python's inspection capabilities to extract the AST for the HECO block at run-time, and then translates it to our high-level IR. This is straight-forward for self-contained code. However, developers can call arbitrary functions in their code. While we could follow the same approach to extract and translate those, too, this would cause the FHE program to explode in size. In addition, many popular Python func-

tionalities are backed by precompiled C implementations, for which one would not be able to extract an AST either way. Here, we exploit the data-independent nature of FHE and execute black-box functions on generic *tracing-style* inputs with overloaded operators, and use the resulting expression tree to inline the function call in the generated IR.

High-Level IR. Our high-level IR represents the application logic of the program, not its eventual FHE implementation. Therefore, at this level, operations do not need to correspond to specific native FHE instructions. Our IR essentially represents standard imperative programming, and supports high-level control-flow instructions (loops, If/Else, etc.), functions, a rich type system, and other features to make expressing high-level programs straightforward. At the same time, we keep complexity low by not considering aspects of high-level language that do not impact FHE code generation. For example, we do not differentiate between stack- and heap-allocated variables, allowing us to focus on the semantics of the computation rather than low-level details of memory allocation which will become meaningless once the program is translated into FHE.

4.3 Translation & Optimization

HECO's middle-end optimizes the high-level IR program, mapping it from the traditional imperative paradigm to FHE's unique programming paradigm. Our optimizations and conversions are implemented as independent passes, which allows us to dynamically compose compilation pipelines for a specific application and target scheme. This modular approach also makes it easy to extend our system without having to modify HECO's code base. External tools can be easily integrated by emitting the current IR representation and round-tripping through the external tool before reading in the externally modified IR. We focus primarily on transformations that can only be performed efficiently while we still have access to the semantic information (e.g. types, control structures, etc) of the high-level language. This stage includes standard compiler optimizations like CSE and/or variable substitution, constant folding, in-lining, and loop fusion/unrolling, which in turn enable our FHE-specific optimizations like our automated SIMD batching to apply more generally. In Section 5 we present out new automatic batching optimizations in details.

Our system progressively lowers the program to our HE IR, the operations in which map directly to native FHE operations. This lowering does not need to occur all at once, and our high-level IR and HE IR can be combined to represent partially lowered programs. HECO's HE IR follows the pattern set out by the proposed FHE Evaluator API [5]. In addition, it provides support for optimized versions of instructions (e.g., `he.square` in addition to `he.mult`) that some backends offer. A program in pure HE IR is equivalent to a circuit, and we can apply the various existing circuit optimizations at this level. For example, inserting relinearization and other ciphertext maintenance operations. We omit a detailed description of these techniques here, as they are not the focus of this paper.

4.4 Code Generation & Back-End

Since current FHE libraries expect to be used directly by developers writing code that will be linked to them, they have no concept of accepting a circuit for evaluation or executing a pre-compiled program. Existing tools tend to provide a thin abstraction layer on top of these libraries, e.g., EVA includes such a *runtime system* backed by SEAL. However, this makes it difficult to analyze or debug the output of those tools, since they usually have no way to convert the program into a human-readable format prior to execution. HECO instead emits `C++` source code implementing the program against the target library. This allows developers to inspect the generated code much more easily, and enables integration of HECO into existing toolchains. Note that this is just one possible code generation target, and it would also be trivial to instead produce ready-to-run executables. However, this shifts the burden of dealing with issues such as cross-compilation into HECO, and we believe that our current approach offers better separation of concerns and a better user experience. Beyond targeting libraries, HECO is designed to be extensible to support generating code for future FHE-specific hardware accelerators [14].

5 Automatic SIMD Batching

In HECO we introduce a new FHE-specific optimizations that automatically rewrite standard imperative programs to better match the unique programming paradigm of FHE. Our optimization harness the performance capacity SIMD parallelism present in most modern schemes by automatically identifying and exploiting opportunities to apply batching. Due to the large capacity of FHE ciphertext (usually 2^{13}–2^{16} slots), effective use of batching is arguably the single most important optimization for many applications, drastically reducing ciphertext expansion overhead and computation time.

Enabling non-experts to write efficient and secure FHE applications therefore requires abstracting away the complexities of this FHE programing paradigm. Developers should be able to express their algorithms comfortably in the standard imperative paradigm, making use of e.g., loops that access and manipulate vector elements by accessing individual indices. However, such programs do not align well with the restricted set of homomorphic SIMD operations and rotations offered by FHE schemes. Therefore, we need an automated way to transform such programs to the FHE paradigm. While it is possible to emulate index accesses into a vector using a combination of rotations and multiplications with masks, this approach is highly impractical as both rotations and multiplications are computationally expensive.

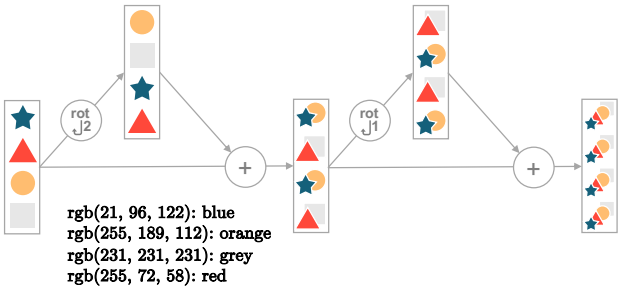

Figure 2: TODO: say somthing here

Instead, we must find a way to translate programs to FHE that preserves and exploits the parallelism present in those programs. This means moving from a program manipulating individual vector elements to one operating over entire vectors in a SIMD fashion. Traditional vectorization algorithm generally rely on the ability to efficiently scatter/gather elements into and out of vectors. However, in FHE the only efficient data movement operation is a cyclical rotation of elements within a vector, realized via automorphisms. Since they are computationally expensive, we want to minimize the amount of rotation and masking operations.

In the following, we describe the core idea behind our automatic transformation, and then walk through the individual challenges and our solutions on how to efficiently identify and exploit these bathing opportunities while keeping compile times minimal.

Core Idea. HECO translates operations over vector variables that are expressed using index-accesses into efficient combinations of FHE-native SIMD operations and cyclical rotations over entire vectors. We encode/encrypt input vectors into the ciphertext slots continuously, i.e., a fresh ciphertext encrypting a vector `x` contains `x[i]` in slot *i*. For scalar inputs, we choose to encode/encrypt their value into all slots of a ciphertext, rather than batching all scalar inputs into a single ciphertext. While this introduces some redundancy, it avoids the need for expensive rotations in purely scalar code and enables efficient vector-scalar operations. Using this approach, each (input) variable corresponds to a ciphertext, and in the following we will frequently refer to variables in a program and the ciphertexts encrypting them interchangeably.

Single Instruction, Multiple Data (SIMD) operations, by definition, work element-wise, applying the same operation to each slot of the involved ciphertexts. In practice, not all elements in a vector can necessarily be computed using the same set of operations. In addition, elements with different indices from the same or different vectors might need to interact. Using rotations, we can bring any arbitrary ciphertext *slot* (i.e., vector index) to a position where it can interact with another arbitrary slot or index in another ciphertext. By multiplying with a (plaintext) mask vector consisting of 0's and 1's, we can select only certain slots from a result and add them to another (masked) ciphertext to express computations beyond the pure element-wise SIMD operations. However, since these combinations are expensive, we want to maximize the amount of computation we can perform using only SIMD operations and minimize the number of rotation and masking operations that are required.

HECO translates programs into efficient batched representations by collecting information on the pattern of index accesses in the program, collecting as many potentially parallelizable operations as possible before a data-dependency or end-of-scope forces the vector to be combined through masking, rotations and addition. Doing so efficiently requires solving a variety of challenges, as a naive approach that simply collects statements affecting vector indices is neither efficient nor does it lend itself to solving the underlying optimization task. HECO considers the context in which index accesses appear in the program, and uses this to build an understanding of which indices from which vectors interact with each other and of the operations applied to each slot of a vector, tracking which slots are "alike", i.e., can be computed at the same time using SIMD operations. In combination, these allow us to efficiently determine the minimal set of rotations and operations required to realize a computation.

Rotation Tracking. In order to identify batching opportunities, we must understand how different vector elements interact. In a standard (loop-heavy) program vectors are accessed at hundreds or thousands of unique indices. Storing this directly as (vector, index) pairs would be highly inefficient. Instead, we leverage the insight that only the *relative* offset between two elements is relevant.

For example, to compute `x[0]+y[2]` we need to rotate either `x` by $+2$ slots or `y` by -2 slots, and the same holds for `x[5]+y[7]`, allowing us to minimize the amount of data we need to store. In many situations, including our example above, there are multiple ways to align the slots that need to interact with each other. Storing all of these and later trying to determine the best sequence quickly leads to a combinatorial explosion. Instead, we consider the wider context, e.g., `z[7]=x[5]+y[7]`, to identify a *target slot*. In our example, the assignment to `z[7]` uniquely determines (`x`, $+2$) as the only appropriate rotation. This allows us to determine a unique set of (relative) offsets (and corresponding rotations) that allows all variables to interact and the result to be stored in the target slot.

Expression Tracking. Rotating vectors so that the required elements can interact ensures that the correct result is written to slot *i*, but also applies the same SIMD perations to all other elements. Applying masking each time to prevent this would be highly inefficient, negating the benefits of batching. Instead, HECO tracks the stored expressions lazily and only

applies masking/combinations when a result needs to be realized (e.g., data-dependency or end of scope). Tracking the state of each vector element through straight-forward variable substitution would be prohibitively expensive, requiring us to track as many expressions as their are slots. Instead, HECO exploits our insights into rotations and relative offsets to compress the data we track by removing redundant information. We transform each expression from a scalar-valued operation over individual elements into an operation over entire ciphertexts, e.g. we replace `x[5]+y[7]` by $x_{-2}+y$ where x_{-2} refers to a copy of `x` that has been rotated by -2 slots. This allows us to track a single expression for all target slots that require the same relative offsets, drastically reducing the storage complexity and laying the groundwork for batching code generation.

Batching. HECO processes the statements in the program sequentially, applying the standard transformation we discussed at the beginning of this section. For each statement that interacts with vectors, we update the set of required rotations and the required expression for vector slots. Afterwards, we eliminate the statement from the program. This is possible, since our tracking data-structures contain all the information required to reconstruct the operations, and because they are designed to integrate with the entire program transformation pass. For example, if a scalar variable is assigned to one slot, updated and later assigned to another slot, correctness is preserved and the first slot will refer to the old state of the variable, while the second slot will use the updated value.

Eventually, this internal state must be converted back to code and re-introduced into the program. However, since batching optimizations work best when we can apply operations to many inputs at the same time, we want to keep collecting information and removing statements for as long as possible. Therefore, HECO applies ... and aggressively inlines variables and expressions to delay having to emit code until, e.g., the end of a scope. However, if the tracking overhead grows too large, we will emit code earlier and re-start tracking, in order to ensure efficient compilation even for extremely large programs.

Naturally, when emitting code, we do not want to simply re-generate the same statements. Instead, HECO generates efficient batched code that exploits the SIMD properties of the FHE scheme. Due to the design choices in our rotation and expression tracking, generating batching-based code from the internal representation is efficient and comparatively straightforward. For each variable that our data-flow analysis indicates needs to be updated, we first collect the required rotations occurring in the expressions.

Once we have emitted the code to realize the required rotations, we emit a single statement for each unique expression occurring in the vector. Since we represent expressions with relative offsets, this might be just a single expression (and therefore statement) for the entire vector. These statements

```
int fold(secret int[] x) {
    int sum = 0;
    for (int i = 0; i < n; i++)
        sum = sum + x[i];
    return sum;
}
```

Listing 1: Loop computing the sum over a vector

compute the expression over the entire vector using SIMD operations. If necessary, we insert multiplications of these intermediate results with a plaintext mask vector of 0's and 1's that will zero out all slots except for those where the final vector should actually contain this expression. Finally, if there is more than one, we emit code to add all intermediate results together to form the final result vector.

Handling Self-Interaction In addition to handling interaction between different ciphertexts, we also need to handle settings where elements of the same vector are interacting with each other. This occurs, for example, in *fold*-style loops as seen in Listing 1. Note that there exists a data-dependency between each iteration of the loop, which will force a naive solution to realize each assignment to `sum`, requiring a series of expensive rotation and masking operations each time. Instead of requiring $O(n)$ rotations, this can be achieved in $O(\log n)$ operations using a *sum-and-rotate* approach. This exploits the cyclical nature of rotations, using a series of copies and rotations to *fold* a vector onto itself, until eventually all slots contain the sum of all vector elements, as can be seen in Figure 2. HECO supports generic rotation-based folding operations and automatically detect opportunities to apply them. This is realized through aggressive variable substitution and a generalization of (commutative) binary expressions (e.g., $+$, $*$) to *n-ary expressions*. This allows each iteration of the loop to simply extend the n-ary expression's operand list, with the final realization of this computation to rotations being delayed until the target variable needs to be realized.

6 Evaluation

In this section, we demonstrate the significant speedups that HECO's automated batching can introduce. We compare naive non-expert implementations against ou automatically optimized versions for a number of example applications. In addition, we compare HECO against the advanced synthesis-based batching approaches created by Porcurpine [**?**], a tool that aims to improve batching beyond what even experts usually achieve. Note that Porcupine fills a very different niche, being more suitable for experts since it requires the developer to provide a sketch of the batched program, and then uses expensive synthesis approaches to explore the space of

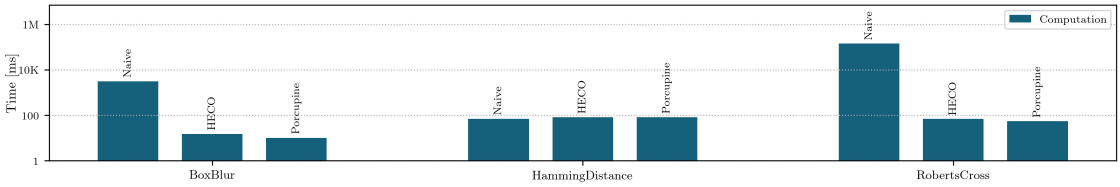

Figure 3: Runtime of example applications in ms, for a naive non-batched solution, our system and Porcupine.

all possible batching programs. While this incurs very high compilation times (up to dozens of minutes) and limits it to small programs with only a handful of operators. Nevertheless, since it does occasionally improve upon the standard 'folklore' expert solutions, we consider it a useful comparsion in order to provide a second data point on the opposite end of the spectrum of the naive implementation.

Implementation HECO is implemented in C++ using the LLVM/MLIR compiler framework [28], which allows us to represent ASTs, high- and low-level IRs in the same framework. In its current realization, 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 and CKKS scheme, with a majority of the API being common to both.

Evaluation We show four example applications (BoxBlur, HammingDistance, RobertsCross) which represent a variety of vector-based computations, primarily from the domain of image processing. We select these because they each allow significant batching opportunities which are, however, non-obvious when expressed in the imperative paradigm. In Figure 3, we show the runtime for the encrypted programs, using SEAL Default Parameters to enable a fair comparison between the different approaches. As we can see, the difference between naive appraoches and HECO's generated batched code is significant. While the differences between HECO and the optimal Porcupine solution are noticeable, they are negligible in comparison with the naive version.

7 Related Work

In this section, we briefly discuss related work in the domain of secure computation compilers, focusing primarily on FHE compilers. While Multi-Party Computation (MPC) compilers [23] face some similar challenges, e.g., the cost of additions being negligible compared to that of multiplications, the MPC programming paradigm is fundamentally different (e.g., focus on communication cost, ability to branch, no noise). A recent survey [35] of FHE compilers found that they fail to fully address the challenges, because existing tools tend to focus on single problems in isolation.

The EVA compiler [15] is likely the most practical existing FHE compiler, offering a user-friendly high-level interface and near-optimally inserting ciphertext maintenance operations into the circuit. This significantly improves the developer experiences, making it accessible to non-experts and reducing time to solution for experts. EVA only supports the CKKS scheme and requires developers to manually map their program to the FHE programming paradigm. Therefore, we consider our work and EVA near orthogonal. In the future, we can envision a modular toolchain that starts by using HECO to translate an imperative input program to the FHE paradigm and then uses EVA's ciphertext-maintenance insertion to optimize it further.

The porcupine compiler [13] 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 trying to outperform the traditional 'folklore' techniques used by experts. Porcupine requires that developers provide a sketch describing the structure of the batched program, making it less suitable for non-expert users. Since it explores a large state space in the search for an optimal solution, compile times tend to be immense (up to many minutes) and programs can contain at most a handful of statements before the approach becomes infeasible.

7.1 Binary Emulation Based Tools

Tools like Cingulata [6], E3 [9] or Google's Transpiler [21] translate their input programs into binary circuits, rather than arithmetic circuits. This allows them to support a wider ranger of operations, including comparisons and other non-polynomial operations. However, this requires bit-slicing integer inputs, i.e., encrypting each bit of an input separately. In this representation, basic arithmetic operation like addition and multiplication no longer map to individual native FHE operations. Instead, these tools emulate binary arithmetic circuit (adders, multipliers, etc) which take dozens of native FHE operations to perform a single arithmetic operation. As a result, the performance of programs translated using these tools is virtually always non-competitive, frequently taking many orders of magnitude longer than expert-written solu-

tions using integer-based tools. While E3 offers some support for switching from bit-wise to integer computation in an attempt to counteract this issue, the unidirectional nature of this *bridging* means it is frequently inapplicable.

7.2 Domain Specific Tools

Domain specific compilers targeting Machine Learning applications, including CHET [16] and nGraph-HE [1] offer great usability, with e.g., nGraph-HE offering a TensorFlow-based interface. The restricted domain these tools consider allows them to provide a large set of hand-written expert-optimized kernels for common functionality (mostly linear algebra operations). This results in great performance on these workloads, but does not solve the wider issue of FHE accessibility. Since these tools rely on expert-driven pre-determined mappings, rather than automatically identifying optimization opportunities, they do not transfer to other domains or general-purpose tools. In their existing form, they are not extensible or integratabtle, however modular tools based on similar ideas would greatly augment a general-purpose end-2-end toolchain.

References

[1] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir 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, pages 45–56, New York, NY, USA, November 2019. Association for Computing Machinery.

[2] C Boura, N Gama, and M Georgieva. Chimera: A unified framework for B/FV, TFHE and HEAAN fully homomorphic encryption and predictions for deep learning. August 2018.

[3] Zvika Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology – CRYPTO 2012*, volume 7417, pages 868–886. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[4] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory*, 6(3):13:1–13:36, July 2014.

[5] Michael Brenner, Wei Dai, Shai Halevi, Kyoohyung Han, Amir Jalali, Miran Kim, Kim Laine, Alex Malozemoff, Pascal Paillier, Yuriy Polyakov, Kurt Rohloff, Erkay Savaş, and Berk Sunar. A standard api for rlwe-based homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, USA, July 2017.

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

[7] Sergiu Carpov, Nicolas Gama, Mariya Georgieva, and Juan Ramon Troncoso-Pastoriza. Privacy-preserving semi-parallel logistic regression training with fully homomorphic encryption. *BMC medical genomics*, 13(Suppl 7):88, 21 July 2020.

[8] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*, volume 10624, pages 409–437. Springer International Publishing, Cham, 2017.

[9] 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*, pages 1–1, 2020.

[10] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. *SpringerLink*, pages 3–33, December 2016.

[11] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *Advances in Cryptology – ASIACRYPT 2017*, pages 377–408. Springer International Publishing, 2017.

[12] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. Technical report, Zama, 15 October 2020.

[13] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T Lee, and Brandon Reagen. Porcupine: A synthesizing compiler for vectorized homomorphic encryption. 19 January 2021.

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

[15] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madanlal 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 December 2019.

[16] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd 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*, pages 142–156, New York, NY, USA, 8 June 2019. ACM.

[17] Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[18] Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, Srini Devadas, Ron Dreslinski, Karim Eldefrawy, Nicholas Genise, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption (extended version). 11 September 2021.

[19] Craig Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the 41st Annual ACM Symposium on Symposium on Theory of Computing - STOC '09*, page 169, Bethesda, MD, USA, 2009. ACM Press.

[20] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology – CRYPTO 2013*, volume 8042, pages 75–92. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

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

[22] Shai Halevi and Victor Shoup. Faster Homomorphic Linear Transformations in HElib. In *Advances in Cryptology – CRYPTO 2018*, volume 10991, pages 93–120. Springer International Publishing, Cham, 2018.

[23] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure Multi-Party computation. In *IEEE Symposium on Security and Privacy (SP)*, pages 479–496, Los Alamitos, CA, USA, 2019. IEEE Computer Society.

[24] Ilia Iliashenko. *Optimisations of Fully Homomorphic Encryption*. PhD thesis, PhD thesis, KU Leuven, 2019.

[25] Wen jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. Pegasus: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. Cryptology ePrint Archive, Report 2020/1606, 2020. https://eprint.iacr.org/2020/1606.

[26] Sreekanth Kannepalli, Kim Laine, and Radames Cruz Moreno. Password monitor: Safeguarding passwords in microsoft edge. https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft- 21 January 2021. Accessed: 2021-7-5.

[27] Miran Kim, Arif Harmanci, Jean-Philippe Bossuat, Sergiu Carpov, Jung Hee Cheon, Ilaria Chillotti, Wonhee Cho, David Froelicher, Nicolas Gama, Mariya Georgieva, Seungwan Hong, Jean-Pierre Hubaux, Duhyeong Kim, Kristin Lauter, Yiping Ma, Lucila Ohno-Machado, Heidi Sofia, Yongha Son, Yongsoo Song, Juan Troncoso-Pastoriza, and Xiaoqian Jiang. Ultra-Fast homomorphic encryption models enable secure outsourcing of genotype imputation. 5 July 2020.

[28] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. 25 February 2020.

[29] Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers.

[30] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, Berlin Heidelberg, Berlin, Germany, 2010. Springer.

[31] Pascal Paillier. Public-Key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology — EUROCRYPT '99*, EUROCRYPT, pages 223–238, Prague, Czech Republic, May 1999. Springer, Berlin, Heidelberg.

[32] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):34:1–34:40, September 2009.

[33] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[34] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[35] Alexander Viand. Sok: Fully homomorphic encryption compilers. In *IEEE Symposium on Security and Privacy*, 2021.