Circuit Optimization Using Arithmetic Table Lookups

RAGHAV MALIK, Purdue University, USA
VEDANT PARANJAPE, Purdue University, USA
MILIND KULKARNI, Purdue University, USA

Fully Homomorphic Encryption (FHE) is a cryptographic technique that enables privacy-preserving computation. State-of-the-art Boolean FHE implementations provide a very low-level interface, usually exposing a limited set of Boolean gates that programmers must use to write their FHE applications. This programming model is unnecessarily restrictive: many Boolean FHE schemes support *programmable bootstrapping*, an operation that allows evaluation of an arbitrary fixed-size lookup table. However, most modern FHE compilers are only capable of reasoning about traditional Boolean circuits, and therefore struggle to take full advantage of programmable bootstrapping.

We present COATL, an FHE compiler that makes use of programmable bootstrapping to produce circuits that are smaller and more efficient than their traditional Boolean counterparts. COATL generates circuits using *arithmetic lookup tables*, a novel abstraction we introduce for reasoning about computations in Boolean FHE programs. We demonstrate on a variety of benchmarks that COATL can generate circuits that run up to 1.5× faster than those generated by other state-of-the-art compilation strategies.

1 INTRODUCTION

Fully Homomorphic Encryption, or FHE, is a cryptographic technique that allows evaluating functions directly on ciphertexts without access to a decryption key. FHE is a powerful tool for enabling *secure multiparty computation*, in which multiple mutually distrusting parties collaboratively perform a computation without revealing anything about their private inputs to the other parties.

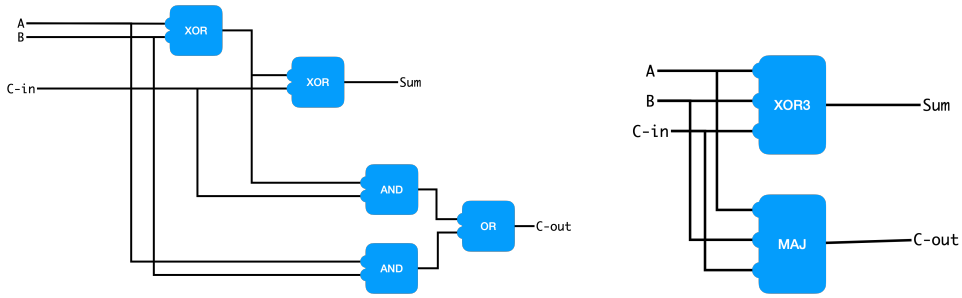
Although FHE presents a promising approach to privacy-preserving computation, writing efficient FHE applications is incredibly difficult: The cryptographic overheads can make computations over secret inputs orders of magnitude more expensive than their plaintext counterparts. Additionally, most state-of-the-art FHE implementations do not offer any high level abstractions for writing FHE applications; programmers are instead burdened with the task of manually mapping their computations down to the library of secure operations the implementation provides. Recent work in this space focuses on developing compilers to automate this process [13, 17, 18, 22, 28, 31].

1.1 Boolean FHE Compilers

Boolean FHE schemes are schemes in which ciphertexts can be thought of as encryptions of *bits* rather than integers (Section 2.1 explores this distinction). Libraries that implement Boolean FHE schemes often also provide efficient implementations of a handful of Boolean gates, making them popular targets for compilation, because existing Boolean circuit synthesis techniques can be used to automatically translate high-level programs into circuits made up of these gates. However, these compilation techniques often fail to take advantage of the more expressive model of computation actually allowed by many Boolean schemes. In particular, some schemes (such as CGGI, the scheme we use in this paper) support an operation called a *programmable bootstrap* (see Section 2.2), which allows ciphertexts to be used as indices into arbitrary fixed-size lookup tables; Boolean gates can then be encoded in these schemes via their truth tables.

Access to programmable bootstrapping means we are no longer limited to a fixed set of Boolean gates: we can instead build lookup tables that encode more complex functions, and then use these to build smaller circuits. For example, the schematic in Figure 1a uses standard Boolean gates (AND,

Authors' addresses: Raghav Malik, School of Electrical and Computer Engineering, Purdue University, West Lafayette, USA; Vedant Paranjape, School of Electrical and Computer Engineering, Purdue University, West Lafayette, USA; Milind Kulkarni, School of Electrical and Computer Engineering, Purdue University, West Lafayette, USA.



(a) An adder using Boolean gates (b) An adder using custom LUTs

Fig. 1. Circuits can be made smaller by using custom LUTs instead of traditional Boolean gates

OR, and XOR) to implement a full-adder. By instead using custom lookup tables implementing a 3-way XOR and a "majority" operation, we can express the same circuit with only two gates instead of five (Figure 1b). (In fact, the latter circuit corresponds exactly to what an FHE expert might write for a full adder!) Unfortunately, generating this circuit automatically with a compiler proves to be challenging. In general, it is difficult to determine whether a given function can be expressed as a lookup table in this way; indeed, for reasons discussed in Section 4.1, most cannot. Existing circuit synthesis techniques therefore cannot use these custom LUTs when generating circuits.

Our key insight in this paper is that Boolean gates are the *wrong* abstraction to use when programming for a scheme like CGGI. We introduce an alternative abstraction called the *arithmetic lookup table*, which models all CGGI computation as (1) computing a linear combination of a group of ciphertexts followed by (2) using the linear combination to index into a fixed-size lookup table. This abstraction strictly generalizes the old notion of Boolean gates—in particular, a Boolean gate is a special case of an arithmetic lookup table with particular coefficients—but comes with more flexibility and can model a greater class of functions. Armed with this new abstraction, we can now take advantage of the full power of programmable bootstrapping to generate circuits built out of these custom LUTs.

1.2 COATL

In this paper, we present COATL, the first Boolean FHE compiler that uses the arithmetic LUT abstraction to generate circuits with nontrivial custom lookup tables. Rather than synthesizing these circuits from the ground-up, COATL takes *existing Boolean circuits* and identifies groups of gates which can be *merged* into a single custom LUT. The particular contributions we make are:

- We develop and formalize the notion of an *arithmetic lookup table*, which serves as a better model of computation in CGGI than traditional Boolean gates
- We present an algorithm that uses this formalism to determine whether an arbitrary function can be expressed as a custom LUT
- We describe how to shrink an existing Boolean circuit by merging sequences of gates with custom lookup tables.

We implement the above algorithm and transformations in a compiler called COATL. We use COATL to compile a variety of common kernels, and show that it can generate circuits that outperform their naïve Boolean counterparts by up to 1.5×.

2 BACKGROUND

We first give preliminaries on Fully Homomorphic Encryption, and then describe CGGI, the particular FHE scheme used in this paper.

2.1 Fully Homomorphic Encryption

Fully Homomorphic Encryption, or FHE, refers to a class of encryption schemes that allow computations to be performed directly on ciphertexts by an untrusted third party, without revealing the decryption key [15]. A common use case for FHE is *computation offloading*, in which a client encrypts sensitive data and sends the ciphertexts to an untrusted evaluator, along with an *evaluation key*. The evaluator uses the evaluation key to perform some computation over the ciphertexts, and sends the encrypted result back to the client without learning anything about the inputs.

FHE schemes are often classified by how they model ciphertexts. In *arithmetic* schemes, ciphertexts represent encryptions of integers modulo some fixed large prime p. In *boolean* schemes, ciphertexts are usually thought of as encryptions of bits. We describe the scheme used in this paper in more detail in Section 2.2.

Noise Management. Much of the security of FHE schemes comes from injecting a small amount of *noise* into ciphertexts. While the noise level in a freshly encrypted ciphertext is relatively small, performing homomorphic operations causes the noise level to grow, eventually causing decryption to fail. Programmers must therefore be careful to limit the overall depth of their circuits, or manually insert noise management operations called *bootstrapping* [16, 23].

Limitations. While FHE is a powerful technology that enables privacy-preserving computation, its lack of easy programmability prevents it from seeing widespread use. Applications must be programmed manually, and are usually expressed directly as *circuits* that transform ciphertexts using the homomorphic operations provided by the underlying FHE scheme: for arithmetic schemes, this is usually integer addition and multiplication; for boolean schemes, this is a small fixed set of boolean gates.

Furthermore, FHE is *slow*. Even in state-of-the-art FHE implementations, the cryptographic overhead of a single homomorphic operation can be multiple orders of magnitude greater than that of performing the same operation over unencrypted values. Naïvely translating a plaintext function into its ciphertext equivalent can produce circuits that are too expensive to evaluate on any nontrivial inputs. Correctly mapping a computation down into a set of homomorphic operations Writing efficient FHE programs therefore requires a great deal of cryptographic expertise.

2.2 CGGI

This paper uses a scheme called CGGI [23], which supports encrypting n-bit integers for some small n (usually 2 or 3). CGGI natively supports ciphertext addition and scaling ciphertexts by a known plaintext constant. Unlike many of its FHE counterparts, however, it also supports a technique called *programmable bootstrapping*. Unlike a traditional bootstrap, which resets the noise values in a batch of ciphertexts without changing the underlying encrypted values, a programmable bootstrap can only operate on one ciphertext at a time, but additionally evaluates an arbitrary unary function (usually represented as a 2^n-row lookup table) on the encrypted value.

In most implementations of CGGI, these lookup tables are used to capture classic Boolean truth tables: the 2^n-row lookup table can represent a truth table with n inputs. In other words, these lookup tables are abstracted as n-input Boolean gates, with ciphertext inputs treated as bits.

The key observation of this paper is that while this abstraction facilitates easy circuit construction (as various Boolean circuit optimization techniques can be applied), it undersells the flexibility of

CGGI's lookup tables. In actuality, ciphertexts in CGGI are encryptions of integers mod p, where p is the size of the lookup table, and the lookup tables index the result of linear combinations of the ciphertext inputs. (Section 4.1 provides a more formal treatment of this fact.) It is precisely by exploiting this additional flexibility that COATL is able to build more complex lookup tables and create smaller circuits, as we describe next.

2.3 HEIR

MLIR[20] is a compiler infrastructure that aims to simplify the process of writing domain-specific compilers. It allows compiler authors to define "dialects" of custom IR operations, and easily implement "passes" that transform code between these dialects. HEIR[11] is a fork of MLIR that adds a number of FHE-specific dialects and passes, such as a generic `secret` dialect for representing arbitrary homomorphic computation, and several scheme-specific dialects including one for CGGI. HEIR also implements several passes for lowering programs written using the `secret` dialect into scheme-specific operations and generating code for a target FHE implementation. We implement the ideas in this paper on top of the HEIR infrastructure, and in particular, on top of an existing pipeline of passes that transform `secret` code into circuits built out of 2- to 3-input Boolean gates, lower these gates into CGGI operations, and then generate code for the OpenFHE library [3]. This pipeline performs some optimizations on the Boolean circuit before lowering to CGGI, but these optimizations stay in the realm of Boolean gates, and do not take advantage of any of the CGGI-specific techniques we discuss here.

3 OVERVIEW

The basic unit of computation in CGGI is the eight-row[1] lookup table (LUT): A table of possible outputs is used to represent a simple function, and the inputs are combined and used to index into this table. The output can then be used as the input to another table; by combining LUTs in this manner, we can compute more complex functions.

As mentioned in Section 2.2, CGGI programs are typically expressed as Boolean circuits where the lookup tables correspond to Boolean gates. An eight-row LUT is enough to encode any three-input Boolean gate, by indexing into the table with a three-bit integer comprised of the inputs to the Boolean gate. However, as discussed in Section 2.2, p-row lookup tables work by computing a linear combination of the inputs (in $\mathbb{Z}_p$), and using the result to select the appropriate output. Thus, it is often possible to represent much larger gates by exploiting the integer, rather than Boolean, nature of the computation. For example, we can encode a *7-way* AND by summing all the inputs into a single integer between 0 and 7, and indexing into a table with a 1 only in the 7th row. Intuitively, this encoding exploits the fact that the AND gate is invariant under permutations of its inputs, obviating the need to perfectly distinguish each input, and instead only check whether all of the inputs are 1.

COATL's approach relies on the following insight: a Boolean function that is invariant under symmetries of its inputs can often be "compressed" as above. This compression means that the lookup tables can be used to encode gates with larger fan-ins, resulting in circuits that use fewer gates overall. We call these more general LUTs *arithmetic lookup tables*, since they are allowed to compute arbitrary linear combinations on their inputs before indexing, unlike traditional CGGI LUTs which always use power-of-two coefficients.

[1]The standard set of encryption parameters in the literature yield eight-row lookup tables [23], though bigger tables are achievable via significantly more expensive parameters. This paper continues with the convention of eight-row lookup tables, but its ideas are applicable to other parameter sets and table sizes as well.

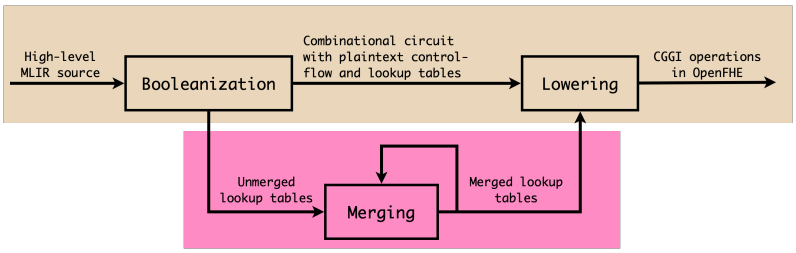


Fig. 2. Overview of boolean FHE workflow. The red highlight denotes COATL's workflow

In this section, we first describe at a high level the usual workflow for compiling Boolean FHE programs, and then give an overview of the compilation strategy COATL uses for generating circuits with arithmetic LUTs. This consists of two "standard" phases that COATL implements (*Booleanization* and *Lowering*) and a new *Merging* phase that compresses the circuit using arithmetic LUTs. These workflows are illustrated in Figure 2.

3.1 Compiling Boolean FHE

A common strategy for compiling Boolean FHE programs—for example, as implemented in HEIR and the TFHE Transpiler [11, 17]—consists of two phases: **Booleanization**, and **Lowering**. COATL uses these phases as well, so we discuss each of them at a high level, using the snippet in Figure 3a, which adds two encrypted 8-bit integers, as a running example. Note that for readability, Figure 3 (and the other code snippets in this paper) are presented using a simplified syntax that nevertheless corresponds structurally to the actual frontend MLIR we use (see Section 5). In particular, homomorphic operations are wrapped in a `secret` block that explicitly captures ciphertext variables, operates on them as plaintexts, and "yields" the result back as a ciphertext.

Booleanization. The aim of Booleanization is to convert a high-level program into a Boolean circuit. This transformation preserves high-level plaintext control-flow such as conditional branches and loops, but converts ciphertext-dependent branches into multiplexed circuits, as is standard [1, 17, 25][2]. Within each (plaintext-dependent) basic block, all encrypted integers are turned into arrays of encrypted bits, and all supported[3] operations are converted to their fixed-bitwidth Boolean counterparts. Finally, we invoke Yosys [30], an open-source circuit synthesis suite, which performs some standard circuit optimizations on each basic block and synthesizes an equivalent circuit built out of 3-input lookup tables[4]. Figure 3b shows the results of Booleanizing the `add_ints` function. Note that the encrypted integral datatype (`enc<i8>`) is implicitly converted to an array of encrypted bits (`enc<i1>[8]`), and the integer addition is converted into a sequence of bitwise operations: `v0` computes the XOR of `x[0]` and `y[0]`, `v1` computes the carry-out, and `v2` computes a three-way XOR between the carry-in, `x[1]`, and `y[1]`.

Lowering. The lowering pass converts an optimized boolean circuit represented via lookup tables into C++ code that invokes cryptographic primitives in our chosen FHE backend, OpenFHE [3]. Similar to Booleanization, this pass preserves all the control flow present in the original circuit,

[2]By default, we choose not to fully unroll every loop with plaintext bounds, as this can sometimes result in very large circuits. However, the programmer can, to some extent, control whether specific loops get unrolled (see Section 5.1).
[3]Currently, the set of supported operations includes integer addition, subtraction, multiplication, common bitwise operations, and comparison.
[4]Yosys is also capable of instead synthesizing the circuit out of standard Boolean logic gates such as AND/OR/XOR/NOT. This usually results in larger circuits with more gates, and therefore worse performance.

```
fn add_ints(x: enc<i8>, y: enc<i8>) -> enc<i8>
{
    let z: enc<i8> = secret(x: i8, y: i8) -> i8 {
        yield x + y;
    };
    return z;
}
```

(a) High-level source

```
fn add_ints(x: enc<i1>[8], y: enc<i1>[8]) -> enc<i1>[8]
{
    let z: enc<i1>[8] = secret(x: i1[8], y: i1[8]) {
        let result: i1[8];
        // v0 = x[0] XOR y[0]
        let v0: i1 = lut(x[0], y[0], 0b0110);
        // v1 = x[0] AND y[0]
        let v1: i1 = lut(x[0], y[0], 0b1000);
        // v2 = x[1] XOR y[1] XOR v1
        let v2: i1 = lut(x[1], y[1], v1, 0b10010110);
        ...
        result[0] = v0;
        result[1] = v2;
        ...
        yield result;
    };
    return z;
}
```

(b) After booleanizing

```
fn add_ints(cc: openfhe.BinContext,
            x: openfhe.ctxt[8],
            y: openfhe.ctxt[8]) -> openfhe.ctxt[8]
{
    result: openfhe.ctxt[8];
    let v0: openfhe.lut = cc.make_lut(6);
    let v1: openfhe.ctxt = cc.mul_const(x[0], 2);
    let v2: openfhe.ctxt = cc.add(y[0], v1);
    let v3: openfhe.ctxt = cc.eval_lut(v2, v0);
    ...
    result[0] = v3;
    ...
    return result;
}
```

(c) Lowered to OpenFHE

Fig. 3. Adding two encrypted 8-bit integers

so the generated C++ may contain branches on plaintext values and loops with plaintext bounds. Figure 3c shows the results of lowering the Booleanized `add_ints` function. Note that a single `lut` operation gets lowered into a sequence of cryptographic primitives that:

(1) Build the lookup table for bootstrapping
(2) Prepare the input to the lookup table by computing `x[0] * 2 + y[0]`
(3) Evaluate the lookup table to bootstrap the input and compute the XOR

3.2 Optimizing Circuits with COATL

As depicted by the red highlight in Figure 2, COATL adds a **Merging** pass to its otherwise standard Boolean FHE compilation pipeline. The goal of this pass is to reduce the total number of lookup tables in the circuit by finding dependent sequences of gates that can be replaced by a single gate with a higher fan-in. If the resulting fan-in is greater than 3, this pass is also responsible for determining the appropriate linear combination to apply to the inputs before indexing into the new lookup table. We save the details of how sequences of gates are identified to be merged, and how the appropriate linear combinations are determined, for Section 4.

In the example in Figure 3b, this pass recognizes that the two gates that compute `v1` and `v2` can be replaced by a single gate that operates on `x[0], y[0], x[1], y[1]` and directly produces `v2`, as shown in Figure 4a. This process is also shown pictorially in Figure 5. In the example, since `v1` has no more uses after merging, it is safe to delete, and hence the total number of gates in the circuit decreases. Section 4 more carefully addresses the general case.

4 DESIGN

The primary motivation behind the optimizations COATL does is to produce circuits with fewer gates[5] that each implement more complex logic. At a high level COATL does so by identifying sequences of computations that can be "merged" into a single gate. The gates that result from merging generally have higher fan-ins, since they compute functions over more inputs simultaneously. *A priori*, a boolean gate with N inputs requires a truth table with 2^N rows, with one row for every possible input configuration. The size of CGGI lookup tables is constrained by the encryption parameters used; our default parameter set yields 8-row ($N = 3$) lookup tables. This presents a problem: How do we encode a gate with more than three inputs using a fixed-size lookup table?

4.1 Arithmetic LUT Formalism

In this section, we develop the notion of an *arithmetic lookup table* (introduced informally in Section 3) to help answer the question above, and give some basic formalisms.

An R-row, n-input arithmetic lookup table consists of the following data:

- A sequence of R boolean outputs: $\mathbf{y} = (y_0, \ldots, y_{R-1})$
- A sequence of n integer coefficients: $\mathbf{a} = (a_0, \ldots, a_{n-1})$

where the coefficients are used to map configurations of n inputs $(x_0, \ldots, x_{n-1})$ to one of the outputs by first computing the *index*:

$$k = \left(\sum_i a_i x_i \right) \bmod R$$

and then returning the corresponding output y_k. Arithmetic LUTs strictly generalize the notion of *truth tables* for boolean gates, like the 3-way OR shown in Figure 6a: Any truth table can be modeled

[5]We distinguish between *gates*, which are abstract units of computation that represent specific functions, and (arithmetic) *lookup tables* (or *arithmetic LUTs*), which are concrete implementations of gates as described in Section 4.1. Similarly, we

```
fn add_ints(x: enc<i1>[8], y: enc<i1>[8]) -> enc<i1>[8]
{
    let z: enc<i1>[8] = secret(x: i1[8], y: i1[8]) {
        let result: i1[8];
        let v0: i1 = arith_lut(
            inputs={x[0], y[0]},
            coeffs={2, 1}, lut=0b0110);
        let v2: i1 = arith_lut(
            inputs={x[1], y[1], x[0], y[0]},
            coeffs={2, 2, 1, 1}, lut=0b01001100);
        ...
        result[0] = v0;
        result[1] = v2;
        ...
        yield result;
    };
    return z;
}
```

(a) The result of merging v1 and v2

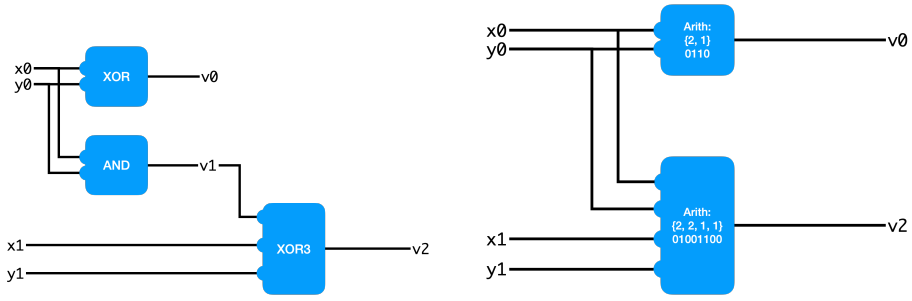
```
fn add_ints(cc: openfhe.BinContext,
            x: openfhe.ctxt[8],
            y: openfhe.ctxt[8]) -> openfhe.ctxt[8]
{
    result: openfhe.ctxt[8];
    ...
    let v4: openfhe.lut = cc.make_lut(76);
    let v5: openfhe.ctxt = cc.add(x[1], y[1]);
    let v6: openfhe.ctxt = cc.mul_const(v5, 2);
    let v7: openfhe.ctxt = cc.add(x[0], y[0]);
    let v8: openfhe.ctxt = cc.add(v2, v3);
    let v9: openfhe.ctxt = cc.eval_lut(v8, v4);
    ...
    result[1] = v9;
    ...
    return result;
}
```

(b) Lowering arithmetic LUTs

Fig. 4. Applying COATL to the booleanized circuit in Figure 3b

as an arithmetic LUT with the same outputs, and the coefficient sequence $\mathbf{a} = \left(2^{n-1}, 2^{n-2}, \ldots, 2^1, 2^0\right)$. In fact, recall from Section 2.2 that this is exactly how the CGGI scheme encodes boolean gates! The expressivity of arithmetic LUTs, however, comes from the ability to choose non-power-of-two coefficients. For example, notice that we can express the same 3-way OR with only four outputs

distinguish between *inputs* (the formal parameters to a gate) and *input configurations* (the sequence of boolean values used in a particular invocation). We use the term *truth table* to refer to a lookup table with power-of-two coefficients.



(a) Schematic of Figure 3c

(b) Schematic of Figure 4b

Fig. 5. COATL merges the `AND` with the `XOR3` to produce a smaller circuit

x	y	z	$x \vee y \vee z$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(a) 3-way OR truth table

x	y	z	$k = x + y + z$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	1
1	0	1	2
1	1	0	2
1	1	1	3

$\mapsto$

k	y_k
0	0
1	1
2	1
3	1

(b) Encoding a 3-way OR into a four-output arithmetic LUT

Fig. 6. Truth tables can be modeled using arithmetic LUTs

instead of eight, by setting $\mathbf{a} = (1, 1, 1)$ and $\mathbf{y} = (0, 1, 1, 1)$ as in Figure 6b. In other words, by cleverly choosing appropriate coefficients, we "compress" the original truth table into fewer rows! This presents a possible solution to the problem posed at the beginning of the section: if a high fan-in merged gate can be expressed as an 8-row arithmetic LUT, we can map it down into CGGI operations.

For a more complex example, consider the truth table in Figure 7 that implements the boolean function "$x_0 \wedge x_1 \implies x_2$". Compressing this into a four-row arithmetic LUT requires (1) partitioning the input configurations of the original truth table, and (2) finding the coefficients for a linear combination that *perfectly distinguishes* the partitions. More precisely:

(1) Any two input configurations in the same partition must also correspond to the same output
(2) Applying the linear combination to two input configurations in the *same* partition should yield the *same* index
(3) Applying the linear combination to two input configurations in *different* partitions should yield *different* indices.

(Note that the second and third conditions mean that we could alternatively think of the coefficients as *determining* a partition, where two configurations are in the same partition if their linear combinations are the same.) The highlighted rows in Figure 7 show one such partition, and the corresponding linear combination $x_0 + x_1 + 3x_2$. Intuitively, this "partitioning-via-compression" strategy works by exploiting *symmetries*. Note that the function is symmetric in its first two inputs:

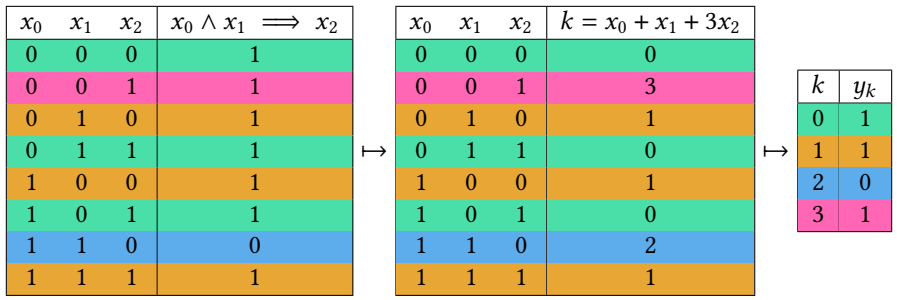
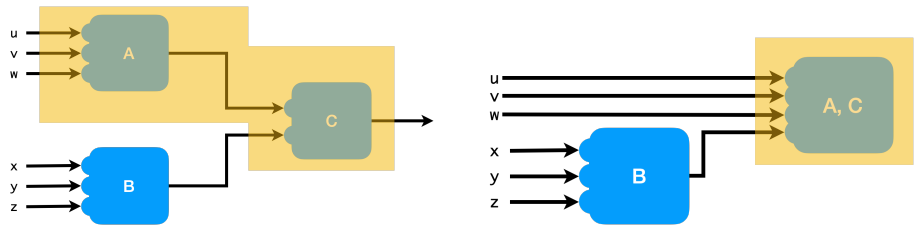


Fig. 7. A truth table can be mapped into a smaller arithmetic LUT by partitioning its rows, and then finding a linear combination that distinguishes the partitions



(a) A dependent pair of gates that can be merged (b) After merging the dependent pair

Fig. 8. Merging can yield circuits with fewer gates

Since we cannot distinguish between (x_0, x_1, x_2) and (x_1, x_0, x_2) we can, for example, place the configurations $(0, 1, 1)$ and $(1, 0, 1)$ in the same partition, and map them to the same output row. More generally, *any set of indistinguishable input configurations can be mapped to the same output row of an arithmetic LUT*, and consequently, highly symmetric gates are more likely to be compressible.

Note that while this example compresses an 8-row truth table into a 4-row arithmetic LUT, with the default parameters COATL can support arithmetic LUTs with up to 8 rows, and can compress functions with up to 7 inputs (i.e. truth tables with up to 128 rows, though not all such truth tables can be compressed). With more inputs per LUT, each arithmetic LUT can compute more complex functions, and result in fewer LUTs in the overall circuit.

4.2 Building Lookup Tables

We can map large truth tables into arithmetic LUTs that fit CGGI's parameters, but how do we actually use these LUTs to compute complex functions? Rather than synthesizing arithmetic LUT circuits from the ground up, COATL makes use of existing infrastructure that maps functions into circuits built out of traditional boolean gates [11, 12, 30], and then finds and merges dependent pairs of gates (in which one gate produces an output consumed by another gate, like A and C in Figure 8a). Consider the example circuit in Figure 8. Merging A and C naïvely produces a four-input gate, but if this merged gate can be compressed into an 8-row arithmetic LUT (as described in Section 4.1), then we can represent the same circuit with only two gates instead of three, as shown in Figure 8b. The remainder of this section describes the procedure for merging a dependent gate pair, and then synthesizing an arithmetic LUT that encodes the merged gate. Section 4.3 discusses how COATL actually identifies pairs to merge.

Algorithm 1: Merging gates into an arithmetic LUT

Algorithm `BuildLUT(g1, g2)`

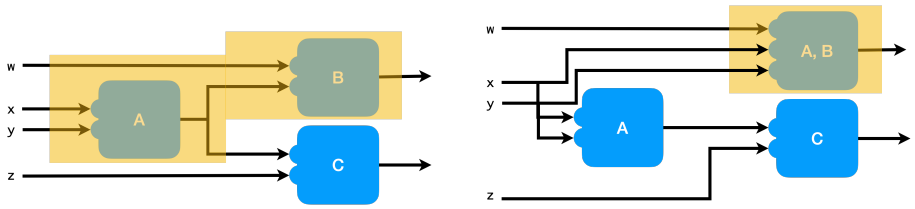
```
dedupInputs ← Canonicalize(g1, g2);
ones, zeros ← Enumerate(g1, g2, dedupInputs);
variables ← {a_i : i ∈ inputs};
constraints ← {−R < a_i < R : a_i ∈ variables};
for o ← ones do
    for z ← zeros do
        constraints.add(∑ a_i o_i ≠ ∑ a_i z_i (mod R));
for o ← ones do
    constraints.add(−R < ∑ a_i o_i < R);
for z ← zeros do
    constraints.add(−R < ∑ a_i z_i < R);
if coefficients ← ILPSolve(constraints, variables) then
    return coefficients;
return error;
```

Given a pair of gates $y_k = g_1(x_1, \ldots, x_n)$ and $z = g_2(y_1, \ldots, y_m)$ in which the output y_k of g_1 appears as one of the inputs to g_2, we want to generate a single (merged) gate that computes $z = g_{1,2}(x_1, \ldots, x_n, y_1, \ldots, \hat{y_k}, \ldots, y_m)$ (where $\hat{y_k}$ means that y_k is omitted from the list of inputs). A priori, the combined gate has $n + m - 1$ inputs and therefore requires a lookup table with 2^{n+m-1} rows to express all possible configurations. Recall from the discussion at the beginning of this section, however, that except for very small values of m and n, such a lookup table is rarely directly expressible with a reasonable set of cryptographic parameters; we need to compress it into an arithmetic LUT with fewer rows. We break the compression process into three steps: *canonicalization*, *enumeration*, and *coefficient synthesis*, described below. Algorithm 1 shows pseudocode for each step.

Canonicalization. Consider the gates $c = g_1(a, b)$ and $e = g_2(a, c, d)$. Naïvely merging these gives $e = g_{1,2}(a, a, b, d)$ which requires a table with $2^4 = 16$ rows. However, since the first two inputs are always identical, the lookup table does not need a row corresponding to, e.g., $(1, 0, 0, 0)$. We avoid using these extra rows by "deduplicating" the set of inputs to produce the smaller $e = g_{1,2}(a, b, d)$. Note that the deduplicated gate explicitly precludes trying to synthesize coefficients that can distinguish between these impossible configurations, thus simplifying the synthesis problem and maximizing the likelihood that synthesis succeeds.

Enumeration. We start by simulating the two gates being merged on all possible configurations to produce a set of 2^{n+m-1} input/output pairs (or fewer, if some inputs were removed in the preceding step), and then we group the configurations based on their corresponding output (either a 0 or a 1). This gives a very coarse partition, which will be refined by the synthesized coefficients in the next step.

Coefficient Synthesis. We query an ILP solver [26] for a sequence of coefficients that determine a partition which (1) refines the coarse partition from the previous step, and (2) maps into the correct number of rows. The ILP problem is formulated as follows:



(a) Before (b) After

Fig. 9. Merging a gate with multiple only shrinks the circuit if all consumers are merged

- Integer variables are created to represent the coefficients for each of the $n + m - 1$ inputs
- Each coefficient is constrained to be within the range $(-R, R)$, where R is the configured number of rows
- For each configuration, the associated linear combination of the coefficients is constrained to be within the range $(-R, R)$, ensuring that each configuration maps to a valid row in an R-row lookup table
- For each pair of configurations with different outputs, the associated linear combinations of coefficients are constrained to be different mod R, ensuring that each row of the compressed lookup table can map to a well-defined output (note that this condition can equivalently be expressed as "two input configurations in the same partition map to the same output")

The solver is configured with a 500 millisecond time limit[6]; if it fails to find a sequence of coefficients within this limit, the two original LUTs are marked as unmergeable. If the solver succeeds, the coefficients are directly used to construct the compressed lookup table as shown in Figure 4a. To reduce the number of expensive solver calls we make, results are cached using the rows of the uncompressed LUT as a key.

4.3 Finding Gates to Merge

Now that we have a mechanism for merging gates to create more complex arithmetic LUTs, we must design a policy that identifies which gates to merge. We begin this discussion by restating two important observations made earlier:

(1) Merging two gates is only possible if the merged gate can be expressed in a fixed-size lookup table (i.e. by exploiting symmetries in the inputs). Thus, intuitively, two gates that compute "simple" functions are easier to merge than two gates that compute "complex" functions.
(2) After merging two gates (a "producer" and a "consumer"), the producer is safe to delete only if it has no other uses remaining. Thus, a merge is "profitable" if and only if the producer can be successfully merged into all of its consumers. For example, in Figure 9, if A and C cannot be merged there is no point in merging A and B. Note that the CGGI semantics and cost model (Section 2) mean that the benefits of a merge *do not depend on the actual gates being merged*, except insofar as they are mergeable.

COATL uses a search heuristic that attempts to maximize the number of "mergeable" gates at every iteration, where a gate is considered mergeable if it can be successfully merged with all of its consumers (and consequently deleted after merging). Because actually attempting to perform a merge involves an expensive call to an ILP solver (see Section 4.2), we instead use a "complexity

[6]This time limit is safe to set: a spurious UNSAT result affects the efficiency, but not the correctness, of the generated code. None of our benchmarks run into the 500 ms limit.

Algorithm 2: Merging gates in a boolean circuit

```
Algorithm MergeGates(circuit)
    candidates ← {g ∈ circuit.gates : Consumers(g) ⊆ circuit.gates};
    while candidates ≠ ∅ do
        next ← arg min_{g∈candidates} IncreaseInComplexity(g);
        candidates = candidates \ {next};
        mergedLuts ← ∅;
        for c ← Consumers(next) do
            lut ← BuildLUT(next, c);
            if lut = error then
                Go to next candidate;
            mergedLuts.add(lut);
        PerformMerge(next, Consumers(next), mergedLuts);
Procedure IncreaseInComplexity(gate, consumers)
    old ← Σ_{c∈consumers} Complexity(c.inputs);
    new ← Σ_{c∈consumers} Complexity(c.inputs \ gate.output ∪ gate.inputs);
    return new − old
```

score" based on input arity as a proxy[7]: intuitively, the fewer inputs a gate has, the more likely it is to be expressible as a LUT. In each iteration, COATL first builds a set of candidates consisting of gates whose outputs are only consumed by other gates (and thus can be merged and then deleted). It then iterates over the candidate set to find the gate that minimizes the overall increase in complexity after merging with all its consumers, and then attempts to perform all the merges. If any of the merges are unsuccessful, they are all rolled back and the gate is removed from the candidate set for the next iteration. Pseudocode for this procedure is shown in Algorithm 2. Note that the `CanBuildLUT` procedure call implements the steps described in Section 4.2 (in particular, the call to the ILP solver), and `PerformMerge` actually updates all the consumers and deletes the now-unused producer.

5 IMPLEMENTATION

COATL is implemented on top of HEIR[11], an MLIR fork that adds various FHE-specific dialects and passes. The core of the implementation is a single `merge-luts` pass that performs the iterative search-and-merge transforms laid out in Section 4. This section discusses details of the rest of the implementation; in particular, we describe how we deal with unrolling loops, and our code generation strategy.

5.1 Unrolling Secret Loops

Recall from Section 3.1 that while COATL supports programs with plaintext-dependent control flow, booleanization happens *at the level of basic blocks*, since boolean circuits do not have a notion of branching-looping control flow. The programming model for HEIR involves enclosing regions of computation inside `secret` blocks to indicate that the operations therein must be booleanized;

[7]In our implementation, gates with between 0 and 3 inputs are "free", since they can trivially be expressed with an 8-row table. Gates with up to 5 inputs contribute 1 point to the complexity score, and gates with more than 5 inputs contribute 2 points.

```
fn sum(nums: enc<i8>[10]) -> enc<i8> {
    let mut acc: enc<i8> = nums[0];
    for (u32 i = 1; i < 10; i++) {
        acc = secret(nums: i8[10]) {
            let new_acc: i8 = acc + nums[i];
            yield new_acc;
        };
    }
    return acc;
}
```

(a) Loop is not unrolled

```
fn sum(nums: enc<i8>[10]) -> enc<i8> {
    let ans: enc<i8> = secret(nums: i8[10]) {
        let mut acc: i8 = nums[0];
        for (u32 i = 1; i < 10; i++) {
            acc = acc + nums[i];
        }
        yield acc;
    };
    return ans;
}
```

(b) Loop is unrolled

Fig. 10. By changing the placement of the `secret` block, the programmer can control whether or not the loop gets unrolled.

thus, a `secret` block cannot contain loops. We implement a pass which runs before booleanization and ensures this is true by fully unrolling all loops contained inside `secret` blocks.

Note that by carefully placing `secret` blocks, the programmer can control which loops get unrolled: Compare the snippets in Figure 10: by placing the `secret` block *inside* the loop body in Figure 10a, the programmer ensures that each iteration of the loop is booleanized separately, and the loop appears in the final generated code. By contrast, since the `secret` block wraps the entire loop in Figure 10b, it is fully unrolled and booleanized into a single circuit that computes every iteration[8].

5.2 OpenFHE Code Generation

COATL targets the OpenFHE [3] backend, a C++ library containing an efficient implementation of the CGGI scheme. Merged lookup tables are converted into OpenFHE-specific code by first translating each lookup table into its associated CGGI operations, and then lowering these operations to HEIR's OpenFHE dialect. Multidimensional arrays are quite common in the final OpenFHE IR (for instance, encrypted integers are translated into arrays of encrypted bits, and arrays of integers therefore become two-dimensional arrays), so the lowering often produces high-level array operations that use broadcasting semantics. We manually implement these broadcast semantics,

[8]We prepared an experiment to investigate the effects of different unrolling choices on efficiency, but the rolled-loop benchmark triggered an unrelated bug in the HEIR pipeline which prevented us from running it.

and map these high-level operations to our implementation when emitting C++ code. We choose to not directly handle details like encryption and decryption, parameter selection, or key generation: Instead, we simply generate a library that provides efficient implementations of each function found in the original HEIR. The programmer must then write a "harness" that correctly sets up a cryptographic context, encrypts the inputs, and then calls the functions provided by this library.

6 EVALUATION

In this section, we evaluate the effectiveness of COATL. Specifically, we aim to answer the following research questions:

- **RQ1: How do the transformations performed by COATL affect the efficiency of generated code?** We compile a number of benchmarks with both COATL and a baseline compiler which does not include the COATL transformations, and compare the run times of the generated code.
- **RQ2: What impact does COATL have on compilation time?** We measure the time taken to compile each benchmark with and without the COATL pass.
- **RQ3: How well does COATL scale to larger input sizes?** We vary the bitwidths and input sizes of our benchmarks, and assess the impact this has on our speedups over the baseline.

All experiments are performed on a server with AMD Ryzen Threadripper Processor (128 threads) clocked at 2.9 GHz with 252 GB of RAM. To run the benchmark binaries we used numactl to isolate runs on a single core with memory allocation restricted to the same NUMA node.

6.1 How efficient are the programs COATL generates compared to the baseline generated code?

To assesses the performance impact of optimizations that COATL does, we run it on a suite of benchmarks and compare the run time of the optimized circuits with the unoptimized circuits. We do thirty iterations of each benchmark and report the median and speedup of the runtime. For these runs the ILP solver timeout is set to 500 milliseconds. Because there is no standard set of binary FHE benchmarks, we implement several classic arithmetic logic circuits (ADD, MUL) at different bit-widths as well as various algorithms that appear in MPC literature (PIR, PSI) [4–6, 8, 9] at different input sizes:

We evaluate COATL on the following set of benchmarks:

- ADD: A classic boolean addition circuit that adds two integers using a carry-lookahead adder. We evaluate 8-, 16-, and 32-bit adders.
- MUL: A circuit to multiply two integers represented as booleans. We evaluate 8-, 16-, and 32-bit multipliers.
- PIR: Private information retrieval. A classic MPC algorithm that determines whether a (secret) key exists in a set. We evaluate retrieval from 8-way sets.
- PSI: Private set intersection. A classic MPC algorithm that computes the intersection of two (secret) sets. Our implementation intersects sets of cardinality 8 and 16, consisting of 8-bit and 16-bit integers, respectively.

Each benchmark is written in MLIR's `secret` dialect, and lowered to OpenFHE C++ code using HEIR's optimizer tool (opt); The lowered OpenFHE C++ code is then compiled and benchmarked. We compare the run times of COATL- and baseline-generated code in Figure 11. The run time numbers are plotted normalised to the baseline; the absolute run time numbers and the relative speedup is listed in Table 1.

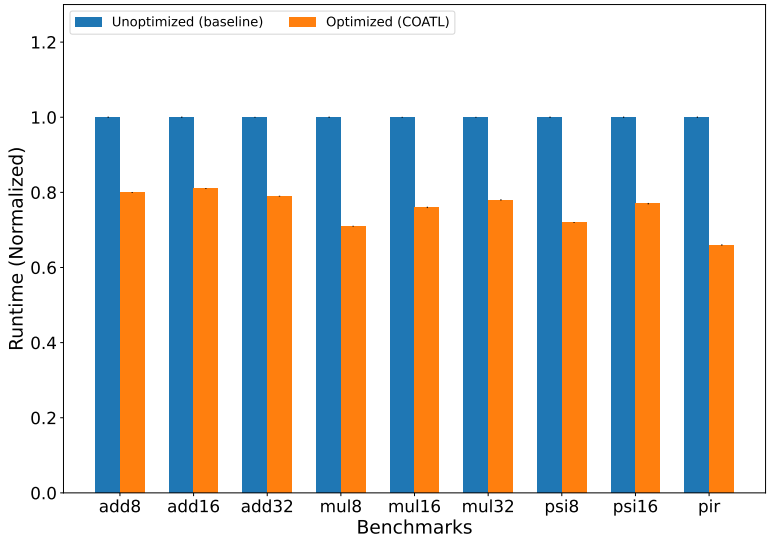


Fig. 11. Baseline vs COATL run times, normalized to baseline. 95% confidence intervals are plotted on the bar graph, but they are miniscule.

Table 1. Runtime statistics of the optimized vs unoptimized benchmarks

Benchmark	Baseline Run Time (ms)	COATL Run Time (ms)	Speedup
add8	331	264	1.25×
add16	684	551	1.24×
add32	1390	1100	1.26×
mul8	1920	1370	1.4×
mul16	8830	6670	1.32×
mul32	35800	27900	1.28×
psi8	25600	18500	1.38×
psi16	244000	187000	1.3×
pir	3960	2620	1.51×

We see that COATL consistently produces more efficient circuits than the baseline HEIR compiler, with speedups ranging from 1.24× to 1.51×. Note that the HEIR implementation already represents an optimized baseline, as the yosys circuit optimizer merges gates—but only using traditional Boolean lookup tables.

We note that COATL's move to arithmetic lookup tables instead of Boolean lookup tables does not result in gates that are more expensive to evaluate. Recall that HEIR's Boolean LUTs still use the same mathematical mechanism as COATL's arithmetic LUTs, but with fixed coefficients. We can see this effect by considering the *circuit size* of our benchmarks, as shown in Table 2. We see that the reduction in gates that COATL achieves is strongly correlated with the improvements in runtime.

Table 2. Gates in the unoptimized vs optimized benchmarks

Benchmark	Baseline gate count	COATL gate count
add8	15	11
add16	31	24
add32	63	50
mul8	87	62
mul16	401	303
mul32	1627	1260
psi8	138	98
psi16	678	526
pir	180	121

6.2 What is the compile time impact of doing the COATL optimization?

Generating optimal code comes with the cost of blow up in the compilation time. Here we compare the compile time of benchmarks and evaluate the time taken by each step as a whole. We use optimization pass time statistics generated by HEIR's optimizer tool (`opt`), which provides compile time information at the granularity of single optimization pass. After `opt` lowers the MLIR to OpenFHE code, we compile it to binaries. The time taken for compiling OpenFHE code is reported as well. For building the binaries we use the same machine and run GNU make with 128 threads.

Table 3. Compile time statistics (in seconds) of unoptimized vs optimized benchmarks. Note that the reported HEIR+COATL time includes solver time

Benchmark	Baseline compilation time (s)			COATL compilation time (s)			
	HEIR	OpenFHE	Total	Solver	HEIR+COATL	OpenFHE	Total
add8	1	3.45	4.45	6	6	3.4	9.4
add16	1	3.83	4.83	14	12.67	3.82	17.82
add32	1	5.16	6.16	26	25.33	5.26	31.26
mul8	1	5.45	6.45	40	39.03	5.32	45.32
mul16	3	34.08	37.08	218	215.45	34.08	252.08
mul32	13	2177	2190	1311	1298	2054	3365
psi8	4	10.6	14.6	47	42.98	9.87	56.87
psi16	22	1301	1323	191	169.27	1186.56	1377.56
pir	2	12.2	14.2	118	116.11	10.61	128.61

Table 3 shows the results. We note a few things. First, we see that across most benchmarks, the time of COATL's optimization pass is dominated by time spent in the ILP solver to determine whether a merge candidate is valid. Second, we see that while COATL time can often be significant, for larger benchmarks (e.g., PSI), compilation time is dominated by the basic process of compiling the resulting circuit, rather than optimization time. Third, for some of the larger benchmarks like psi16 and mul32, the compilation time for compiling the resulting circuit was reduced significantly. This is mainly due to the fact that COATL generates optimized boolean circuits as compared to the baseline.

6.3 How well does the COATL optimization scale to increasing input size?

To evaluate how well the optimization scales to different input sizes, we look at the speedup data in Table 1. The ADD and MUL suite of benchmarks have implemented programs with 8-, 16-, 32-bit input sizes, and the PSI suite has programs with 8-, 16-bit input sizes. As we can see from the data in Table 1, the speedup remains fairly consistent for all the three input sizes for ADD, MUL and PSI suites of benchmarks. In case of ADD, speed up for 8-bit is 1.25, for 16-bit it is 1.24 and for 32-bit it is 1.26. We can roughly say that the speedup stays consistent. But in the case of MUL and PSI, the speedup goes down as we increase the input size. This is due to the higher program complexity of MUL and PSI as compared to ADD.

6.4 Impact of solver timeout on performance and compile time

The HEIR compiler framework ships with a solver [26] which we use to solve the ILP problem described in Section 4.2. We profiled the compile time of the optimization pass and the solver takes the majority percentage of the compile time as shown in Table 3. As a result the compile time impact of COATL increases considerably. The solver can timeout after a specified time limit if it does not find a solution. For the evaluations in Section 6.1 we set the timeout to 500 milliseconds. This timeout is sufficient to avoid missing any optimization opportunities—experimenting with larger timeouts does not produce more-optimized circuits.

7 RELATED WORK

We discuss how COATL relates to other FHE compilers, and to existing circuit optimization techniques.

7.1 FHE Compilers

A large body of work exists in making compilers for FHE [2, 7, 13, 14, 17–19, 22, 28, 31]. Most of these compilers target *arithmetic FHE schemes*, in which the ciphertexts are encryptions of integers modulo some prime p, and the primitive operations are *addition* and *multiplication* modulo p. While arithmetic schemes often have a much lower latency per homomorphic operation and support optimizations such as vectorization, they require more careful ciphertext management, and their inability to compute non-polynomial functions such as comparison make them unsuitable for many applications. In contrast, by targeting a boolean scheme such as CGGI, COATL is able to easily handle a much more general class of programs.

Two of these compilers (Concrete[31], and Google's TFHE Transpiler [17]) instead target boolean schemes, and we discuss them more carefully here. Concrete builds a computation graph from a program written in its Python DSL, and generates code for TFHE [10], a library that implements a boolean FHE scheme similar to CGGI. Concrete performs a number of optimizations on the computation graph, but all of these happen *before* booleanization, and are largely orthogonal to the transformations presented here. In particular, Concrete's compilation workflow also contains a *fusing* pass, which identifies subcomputations that depend on a single ciphertext variable and fuses them into single lookup tables (e.g. replacing a square followed by a sine with a single lookup table that computes $\sin(x^2)$) This fusion differs from COATL's LUT merging in two key ways:

(1) LUT merging is not restricted to subcomputations with a single input
(2) By operating *after* booleanization, COATL merges at a finer granularity, and can merge *across operation boundaries*.

The TFHE Transpiler operates very similarly to COATL: it starts by mapping a high-level program into boolean gates, and then uses standard circuit synthesis techniques like XLS [12] to generate an efficient circuit that can be lowered to TFHE. However, the circuits it synthesizes are restricted

to use the fixed set of boolean gates that TFHE provides, and thus it does not take full advantage of the power of programmable bootstrapping. In particular, the TFHE Transpiler toolchain is roughly equivalent to what we use as a baseline in Section 6.

Gouert et al. [18] propose an Arctyrex, an FHE compiler that also compiles a high level C-like language to boolean FHE. Arctyrex targets a custom implementation of the CGGI scheme designed for GPUs, and does some work to efficiently schedule computations across multiple GPUs. However, it performs very few optimizations on the actual boolean circuit; we expect the techniques presented in this paper to be largely orthogonal to their contributions.

7.2 Circuit Optimization

On the surface, COATL's goals seem similar to existing (non-FHE) circuit optimization techniques, since it attempts to minimize the number of gates in a boolean circuit [12, 24, 30]. However, these techniques cannot, by themselves, capture the unique semantics of CGGI that we exploit here. In particular, the notion of arithmetic LUT that we define here has no counterpart for traditional boolean circuits, as it relies on the fact that CGGI ciphertexts encrypt *integers mod p* rather than actual bits, and can therefore be scaled and added together *as integers*, rather than only combined via power-of-two coefficients.

There is prior work in the space of performing FHE-specific circuit optimizations: Lee et al. [21] show how to use circuit enumeration and program synthesis techniques to learn a set of rewrite rules that can be automatically applied to FHE circuits. They obtain promising results from feeding these rewrite rules into an e-graph [27, 29] and using it to apply local optimizations for, e.g., reducing circuit depth. Unfortunately, their e-graph based approach does not directly work for our use case. E-graphs excel at finding local rewrites; e.g., replacing a self-contained region of gates with a single lookup table. However, recall from our discussion in Section 4.3 that at each step, we have to perform several such replacements at once (i.e, a producer has to be merged into *all of its consumers*); this nonlocal dependence makes it very difficult for an e-graph to reason about individual rewrites.

8 CONCLUSION

This paper presents COATL, a Boolean FHE compiler that takes advantage of programmable bootstrapping in the CGGI scheme. We develop the notion of an arithmetic lookup table, an abstraction that more closely matches the computational model of CGGI, and demonstrate that using this abstraction allows generating circuits that are smaller and more efficient. We demonstrate on a variety of benchmarks that the circuits COATL produces outperform those produced by other compilation strategies by up to 1.5×.

REFERENCES

[1] Jayvant Anantpur and Govindarajan R. 2014. Taming Control Divergence in GPUs through Control Flow Linearization. In *Compiler Construction*, Albert Cohen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 133–153.

[2] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. 2019. RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (London, United Kingdom) *(WAHC'19)*. Association for Computing Machinery, New York, NY, USA, 57–68. https://doi.org/10.1145/3338469.3358945

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

[4] Fattaneh Bayatbabolghani, Marina Blanton, Mehrdad Aliasgari, and Michael T. Goodrich. 2017. Secure Fingerprint Alignment and Matching Protocols. *CoRR* abs/1702.03379 (2017). arXiv:1702.03379 http://arxiv.org/abs/1702.03379

[5] Marina Blanton, Ahreum Kang, and Chen Yuan. 2019. Improved Building Blocks for Secure Multi-Party Computation based on Secret Sharing with Honest Majority. Cryptology ePrint Archive, Paper 2019/718. https://eprint.iacr.org/2019/718 https://eprint.iacr.org/2019/718.

[6] Fabien Boemer, Karl Tarbe, and Rehan Rishi. 2024. Announcing Swift Homomorphic Encryption. https://www.swift.org/blog/announcing-swift-homomorphic-encryption/

[7] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. 2015. Armadillo: A Compilation Chain for Privacy Preserving Applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing* (Singapore, Republic of Singapore) *(SCC '15)*. Association for Computing Machinery, New York, NY, USA, 13–19. https://doi.org/10.1145/2732516.2732520

[8] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. 2018. Labeled PSI from Fully Homomorphic Encryption with Malicious Security. Cryptology ePrint Archive, Paper 2018/787. https://doi.org/10.1145/3243734.3243836

[9] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. Cryptology ePrint Archive, Paper 2017/299. https://eprint.iacr.org/2017/299

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

[11] HEIR Contributors. 2023. HEIR: Homomorphic Encryption Intermediate Representation. https://github.com/google/heir.

[12] XLS Contributors. 2024. XLS: Accelerated HW Synthesis. https://github.com/google/xls.

[13] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. 2021. Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 375–389. https://doi.org/10.1145/3453483.3454050

[14] Eric Crockett, Chris Peikert, and Chad Sharp. 2018. ALCHEMY: A Language and Compiler for Homomorphic Encryption Made EasY. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1020–1037. https://doi.org/10.1145/3243734.3243828

[15] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Ph. D. Dissertation. Stanford, CA, USA. Advisor(s) Boneh, Dan. AAI3382729.

[16] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Better Bootstrapping in Fully Homomorphic Encryption. In *Public Key Cryptography – PKC 2012*, Marc Fischlin, Johannes Buchmann, and Mark Manulis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

[17] Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce Wilson, Asra Ali, Eric P. Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, Phillipp Schoppmann, Sasha Kulankhina, Alain Forget, David Marn, Cameron Tew, Rafael Misoczki, Bernat Guillen, Xinyu Ye, Dennis Kraft, Damien Desfontaines, Aishe Krishnamurthy, Miguel Guevara, Irippuge Milinda Perera, Yurii Sushko, and Bryant Gipson. 2021. A General Purpose Transpiler for Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2021/811. https://eprint.iacr.org/2021/811 https://eprint.iacr.org/2021/811.

[18] Charles Gouert, Vinu Joseph, Steven Dalton, Cedric Augonnet, Michael Garland, and Nektarios Georgios Tsoutsos. 2023. Accelerated Encrypted Execution of General-Purpose Applications. Cryptology ePrint Archive, Paper 2023/641. https://eprint.iacr.org/2023/641 https://eprint.iacr.org/2023/641.

[19] Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. 2024. A Tensor Compiler with Automatic Data Packing for Simple and Efficient Fully Homomorphic Encryption. *Proc. ACM Program.*

Lang. 8, PLDI, Article 152 (June 2024), 25 pages. https://doi.org/10.1145/3656382

[20] Chris Lattner and Jacques Pienaar. 2019. MLIR Primer: A Compiler Infrastructure for the End of Moore's Law.

[21] DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 2020. Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 503–518. https://doi.org/10.1145/3385412.3385996

[22] Raghav Malik, Kabir Sheth, and Milind Kulkarni. 2023. Coyote: A Compiler for Vectorizing Encrypted Arithmetic Circuits. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 118–133. https://doi.org/10.1145/3582016.3582057

[23] Daniele Micciancio and Yuriy Polyakov. 2020. Bootstrapping in FHEW-like Cryptosystems. Cryptology ePrint Archive, Paper 2020/086. https://eprint.iacr.org/2020/086

[24] Alan Mishchenko. 2024. ABC: A System for Sequential Synthesis and Verification. https://people.eecs.berkeley.edu/~alanmi/abc.

[25] Simon Moll and Sebastian Hack. 2018. Partial Control-Flow Linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 543–556. https://doi.org/10.1145/3192366.3192413

[26] Laurent Perron, Frédéric Didier, and Steven Gay. 2023. The CP-SAT-LP Solver. In *29th International Conference on Principles and Practice of Constraint Programming (CP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 280)*, Roland H. C. Yap (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:2. https://doi.org/10.4230/LIPIcs.CP.2023.3

[27] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. *SIGPLAN Not.* 44, 1 (jan 2009), 264–276. https://doi.org/10.1145/1594834.1480915

[28] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. 2023. HECO: Fully Homomorphic Encryption Compiler. arXiv:2202.01649 [cs.CR]

[29] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (jan 2021), 29 pages. https://doi.org/10.1145/3434304

[30] Claire Wolf. [n. d.]. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/.

[31] Zama. 2022. Concrete: TFHE Compiler that converts python programs into FHE equivalent. https://github.com/zama-ai/concrete.