COIL: Compiling Homomorphic Circuits with Control Flow

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

Fully Homomorphic Encryption (FHE) enables evaluating computations over ciphertexts without revealing the encrypted data. A key challenge when compiling for FHE is dealing with control flow: the usual semantics require the party evaluating the program to learn something about each branch, which violates the basic premise of secure computations. Standard "multiplexer"-based compilers solve this problem by generating programs that require the evaluator to execute *every possible path*, obviating the need to know which path is correct. Thus, they provide oblivious semantics at the cost of producing unwieldy circuits that are difficult to effectively optimize.

We present COIL, an FHE compiler that addresses the control flow challenge by restructuring how control flow in circuits is interpreted, replacing the use of multiplexers with *path forests*, and enabling a series of path-dependent optimizations that result in more efficient realizations of complex branching kernels. We demonstrate on a variety of benchmarks that COIL outperforms other state-of-the-art FHE compilation techniques, often by more than an order of magnitude.

1 INTRODUCTION

Fully Homomorphic Encryption (FHE) is a cryptographic technique that allows an evaluator to perform a computation on ciphertexts without learning about its inputs. This enables *secure multiparty computation*, in which two or more mutually distrustful parties collaborate to evaluate a function over their private inputs.

While FHE represents a promising solution to problems like secure machine learning and computation offloading, the overhead of encrypted computation is incredibly high, with FHE programs being orders of magnitude more expensive than their non-homomorphic counterparts. Furthermore, the nonintuitive cost models and limited set of operations supported by most homomorphic schemes make FHE programming both tedious and error-prone. Much of the work in this space therefore focuses on developing compilers to alleviate some of the burden.

1.1 Oblivious Control Flow

One of the key challenges that FHE programmers have to deal with, and the one with which we are most concerned, is the problem of control flow, and in particular, *secure conditionals* in which the condition depends on an encrypted value. The standard control flow semantics would require the evaluator to learn the condition in order to choose the appropriate branch to take. However, FHE semantics prohibit the evaluator from learning the values of *any* ciphertext. Instead, FHE compilers will often transform a program so that the evaluator computes *every possible result*, and then uses the encrypted condition variables and a network of secure multiplexers ("muxes") to obliviously select the correct result at the end (see Section 2.1) [18, 19].

This paper describes an alternative to the "muxing" strategy: we introduce an intermediate representation (IR) called *path forests* (Section 4). The *path forest IR* annotates each possible control flow path through the program with the conditions necessary to witness that path. In other words, the path forest IR keeps track of conditions *throughout* the program instead of only using them at the *end of divergent control flow* like in a mux network. This distinction, while minor, is crucial, as it enables path-dependent optimizations such as *pruning* (removing code for computations that

Authors' addresses: Raghav Malik, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA, malik22@purdue.edu; Milind Kulkarni, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA, milind@purdue.edu.

cannot happen) and *specialization* (instantiating ciphertexts whose values are constrained along each path)[1].

For example, consider the following code snippet which takes two ciphertext inputs, x and y:

```
if (x - 1 < y) {
    z = 1;
} else {
    z = 2;
}
if (x < y + 1) {
    w = z + 3;
} else {
    w = z - 3;
}
```

Naïvely transforming this into a mux network yields[2]:

```
z = mux(x - 1 < y, 1, 2);
w = mux(x < y + 1, z + 3, z - 3)
```

In contrast, the path forest encoding of the above snippet looks[3] like:

```
[x - 1 < y] [x < y + 1] z = 1; w = z + 3;
[x - 1 < y] [x ≥ y + 1] z = 1; w = z - 3;
[x - 1 ≥ y] [x < y + 1] z = 2; w = z + 3;
[x - 1 ≥ y] [x ≥ y + 1] z = 2; w = z - 3;
```

Notice that in the latter encoding, the middle two paths can immediately be identified as unreachable (because of the mutually exclusive conditions), and *pruned* from the forest. Furthermore, while the mux network requires the operations $z + 3$ and $z - 3$ to be done securely since z is a ciphertext, the path forest allows them to be *specialized* to plaintext values and done in the clear since the value of z is known along each path. Note that specializing to plaintext does *not* leak any information to the evaluator, as *every path is still evaluated.* After applying these two optimizations, the path forest becomes:

```
[x - 1 < y] [x < y + 1] z = 1; w = 4;
[x - 1 ≥ y] [x ≥ y + 1] z = 2; w = -1;
```

This can now be converted back into a mux network for evaluation:

```
⟨z, w⟩ = mux(x - 1 < y, ⟨1, 4⟩, ⟨2, -1⟩)
```

However, the path forest encoding enables much more efficient execution strategies. In particular, we show how a path forest can be regarded as a *decision tree*, allowing us to use efficient private decision tree inference techniques [2, 27]. Moreover, the path forest splits the computation into conditions and a set of condition-free subcomputations, allowing the latter to be further optimized through classical techniques such as vectorization [12, 26, 33].

[1] Pruning and specialization can be thought of as versions of dead code elimination and constant propagation that take into account path-dependent information.

[2] The remainder of this paper adopts the convention of underlining ciphertext values in all code snippets.

[3] For ease of presentation, the path forests in the examples are written differently from the formal grammar defined in Figure 5

1.2 Contributions

The specific contributions we make in this paper are:

(1) An IR for FHE programs with branching control flow called *path forests*.
(2) A series of optimizations enabled by the path forest IR, including *specialization* and *pruning*.
(3) Our COIL compiler that uses path forests to optimize branching FHE programs.

We use COIL to compile a set of benchmarks with secure control flow. We demonstrate that COIL greatly outperforms the naïve mux network strategy, often automatically discovering hand-coded optimal implementations of certain operations.

2 FULLY HOMOMORPHIC ENCRYPTION

Fully Homomorphic Encryption refers to a class of encryption schemes that allow for *homomorphic computation* over ciphertexts; e.g., the sum or product of the encryptions of two integers is the encryption of their sum or product. FHE is a useful cryptographic tool for carrying out *privacy-preserving* or *oblivious computation* (evaluating programs where the inputs are unknown to the evaluator).

A common use-case for FHE is *computation offloading*, which uses a single public/private keypair and only involves two parties[4]. The client encrypts their inputs to a program with their private key and sends the ciphertexts to the evaluator. The evaluator uses the public key to evaluate the program via a sequence of homomorphic operations before sending the result ciphertext back to the client, who finally decrypts it and learns the output of the program.

2.1 Oblivious Semantics for Computation

The standard security guarantee for oblivious computation is *noninterference*: an evaluator without the private key cannot distinguish two traces of a program that differ only in encrypted variables. This poses a problem for programs with branching: in knowing which branch to take, the evaluator must learn something about any ciphertext that influences that branch, breaking noninterference. To preserve noninterference we want programs to exhibit *oblivious control-flow semantics*, in which the evaluator can correctly execute a branching program without knowing anything about which branches were taken.

A common way to provide such semantics is via *secure multiplexers* ("muxes"), cryptographic operations that use a ciphertext selector to obliviously choose between multiple inputs, returning the input corresponding to the value of the selector[5]. When the evaluator encounters a branch, it executes *both* paths, and then uses the branching condition and a mux to select the correct value when control flow converges. This obviates the need to reveal anything about the branching condition to the evaluator, and still ensures that the correct return value is produced. Note that this technique can in the worst case have an exponential effect on the total computation time: every path through the program has to be evaluated, even if the result of only one is used.

The use of muxes can be extended to other kinds of secure control flow such as loops: given a plaintext upper bound on the number of iterations, a loop can be fully unrolled into a series of branches that check the exit condition for each iteration, which can be obliviously evaluated as above.

Armed with these constructions, we can build a compiler that achieves oblivious semantics [18, 19]: functions are inlined, loops are fully unrolled, and all the resulting branches are converted

[4]Some authors have proposed multiparty multi-key extensions to FHE [5, 10, 25]. While we do not directly make use of these, the ideas in this paper are relatively straightforward to extend to a multikey setting.
[5]An example of a mux in an FHE scheme that provides homomorphic addition and multiplication is $\underline{b}X + (1 - \underline{b})Y$, where $\underline{b}$ is the (ciphertext) selector bit, and X and Y are the two inputs being selected between.

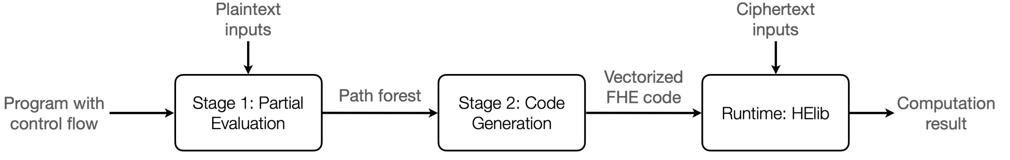


Fig. 1. COIL pipeline

to muxes as described above, resulting in an *arithmetic* or *boolean* circuit, which can be evaluated in any FHE scheme that supports all the operations used.

2.2 Ciphertext Batching

Even without considering the computation-expanding effect of oblivious branching, a major drawback of using FHE is the high overhead of encrypted computation. State-of-the-art implementations of homomorphic operations are often still several orders of magnitude more expensive than their plaintext counterparts. One way to mitigate this overhead is via *ciphertext batching*, an optimization that certain (RLWE-based[6]) FHE schemes provide [8, 31]. Ciphertext batching allows encrypting a vector of integers into a *single* ciphertext in such a way that homomorphic operations occur element-wise on the underlying vector (i.e. SIMD-style).

Ciphertext batching can alleviate the FHE overhead by *vectorizing* computations and reducing the total number of homomorphic operations the evaluator has to perform. For example, a sequence of N independent multiplies can be replaced by a single vector multiply by packing all the left and right operands into ciphertext vectors. These vector semantics have their own limitations that make vectorizing general FHE applications difficult [12, 26, 33].

2.3 Noise Management

Much of the security of FHE schemes comes from a small amount of noise added to each ciphertext upon encryption. A freshly encrypted ciphertext starts with a certain *noise budget*. When the noise level exceeds this budget, it interferes with the encrypted value, causing decryption to fail. Homomorphic operations—in particular, multiplication—accumulate noise; hence, the noise budget roughly corresponds to the *maximum depth* of circuit that can be evaluated. The noise budget can be increased by encrypting into larger ciphertexts, which are in turn much slower to compute over.

Alternatively, some schemes support a technique called *bootstrapping* [16, 17] which "refreshes" the noise budget by homomorphically evaluating the decryption function. Unfortunately, bootstrapping is incredibly slow, and also carries some limitations governing when it can be used. Managing the total depth is, therefore, crucial to designing efficient FHE applications. In Section 6.3, we demonstrate how the path forest strategy can reduce the multiplicative depth of a circuit.

3 COIL OVERVIEW

Our goal is a compiler for FHE programs that contain oblivious control flow. Naïvely compiling control flow into muxes (Section 2.1) often yields poor results: it can obscure opportunities for path-dependent optimization, and can produce expensive circuits that do not to scale. This section outlines an alternative compilation technique for such programs based on *path forests*. A path forest is a representation of all possible control flow paths through a program, where each path is annotated with a sequence of conditions that must be true along it.

[6]RLWE stands for Ring Learning with Errors, a number-theoretic problem that involves distinguishing two distributions of polynomials. The security of FHE schemes such as BFV and BGV (the one used in this paper) is based on the hardness of RLWE.

```
let index = \(haystack, needle, cur) => {
    if (cur < len(haystack)) {
        if (needle == @haystack[cur]) {
            cur
        } else {
            index(haystack, needle, (cur + 1))
        }
    } else {
        len(haystack)
    }
} in
let lookup = \(arr, i, cur) => {
    if (cur == len(arr)) {
        -1
    } else {
        if (i == cur) {
            @arr[cur]
        } else {
            lookup(arr, i, (cur + 1))
        }
    }
} in
let ⟨keys, values⟩ = ⟨ptxts(0, 7), ptxts(8, 15)⟩ in
let key = ctxt(0) in
let idx = index(keys, key, 0) in
lookup(values, idx, 0)
```

Fig. 2. COIL snippet implementing associative array by using the index of a private key to look up a value

At a high level, the COIL compiler first translates the program into a path forest to perform optimizations on it, and then lowers it to FHE primitives (Figure 1). More precisely, COIL is a *staging compiler* with two *stages*:

1. When plaintext inputs become available, they are used to "untangle" the program's control flow into a path forest, resulting in a program that is partially evaluated with respect to the plaintexts (as detailed in Figure 6).
2. The partially evaluated path forest, which now represents only ciphertext computation, is interpreted as a decision tree and lowered to vectorized FHE primitives for decision tree inference such as COPSE [27].

Note that this framing implies that the transformations done by COIL are secure-by-construction: they cannot possibly leak any private information, as they are all performed when only plaintext inputs are available. In fact, the ciphertext inputs only become available at runtime, when they are used to execute the final generated FHE code.

The remainder of this section describes each stage in more detail, and walks through compiling the running example shown in Figure 2, which implements indexing into an associative array with a private (ciphertext) key: the `index` function determines the index of the key in the array, which the `lookup` function uses to retrieve the value[7], returning `-1` if the key is not found.

[7]Note that in the example, since key is a ciphertext, the computed index idx must also be a ciphertext, so we cannot directly index an array with it (Section 4.1).

3.1 Building a Path Forest

The first stage of the compiler builds a path forest by iterating through the following steps:

- Add a path to the forest for each ciphertext-dependent branch.
- Whenever the value of a ciphertext variable can be determined along a particular path, "specialize" that copy of the variable to a plaintext and substitute its value.
- Evaluate any plaintext-dependent branches (i.e. branches that can be determined from public inputs alone) to "prune" away unreachable paths. In particular, this includes branches that have been made plaintext-dependent by the previous step.

Armed with our intuition of the first stage as a partial evaluator, we might expect that applying the procedure above to the example in Figure 2 yields a forest with a single path for each possible value of the key (Figure 3b). Lets see why:

Assume the calls to `get_ptxt_array()` return the arrays `[1; 2; 3]` and `[1; 4; 9]` for `keys` and `values`, respectively. First, call to the `index` function gets specialized to produce:

```
let idx = if (0 < len(keys)) {
    if (key == @keys[0]) {
        0
    } else {
        index(keys, key, 1)
    }
} else {
    len(keys)
} in ...
```

Since both `len(keys)` and `0` are plaintexts, the second branch of the conditional gets pruned away:

```
let idx = if (key == @keys[0]) {
    0
} else {
    index(keys, key, 1)
} in ...
```

and finally, the ciphertext-dependent branch gets converted into the following two paths:

```
[key == keys[0]] idx = 0;
[key != keys[0]] idx = index(keys, key, 1);
```

This process continues recursively for the remaining call to `index`, and similarly for the call to `lookup` in the following line, eventually yielding the path forest shown in Figure 3a.

Specializing on possible values of ciphertexts does not leak any information. For instance, although `key` and `idx` are ciphertexts, their values are uniquely determined on any particular path, allowing them to be replaced by plaintexts along each path. Importantly, using plaintexts for `key` and `idx` does not leak information: Even though each path is evaluated using plaintext values, every path is still evaluated, and the appropriate result is selected securely (Section 5.1).

In a final round of pruning and specialization, the paths that contain conflicting values of `idx` are removed, and the now-plaintext indices into the `values` array are resolved, reducing Figure 3a into the final forest in Figure 3b.

3.2 Path Forests are Decision Trees

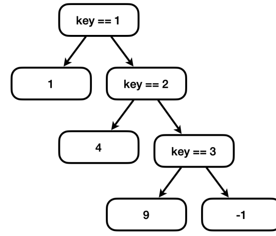
Looking at the path forest in Figure 3b, we notice that the first path can be distinguished from the last three by the condition `key == 1`, the bottom three paths can be further distinguished by the condition `key == 2`, and the last two paths can be distinguished by `key == 3`. The program control

```
[key == keys[0]] idx = 0; [idx == 0] values[0]
                          [idx != 0] [idx == 1] values[1]
                                     [idx != 1] [idx == 2] values[2]
                                                [idx != 2] -1
[key != keys[0]] [key == keys[1]] idx = 1; [idx == 0] values[0]
                                           [idx != 0] [idx == 1] values[1]
                                                      [idx != 1] [idx == 2] values[2]
                                                                 [idx != 2] -1
[key != keys[0]] [key != keys[1]] [key == keys[2]] idx = 2; [idx == 0] values[0]
                                                            [idx != 0] [idx == 1] values[1]
                                                                       [idx != 1] [idx == 2] values[2]
                                                                                  [idx != 2] -1
[key != keys[0]] [key != keys[1]] [key != keys[2]] idx = 3; [idx == 0] values[0]
                                                            [idx != 0] [idx == 1] values[1]
                                                                       [idx != 1] [idx == 2] values[2]
                                                                                  [idx != 2] -1
```

(a) Extracted path forest

```
[key == 1] idx = 0; 1
[key != 1] [key == 2] idx = 1; 4
[key != 1] [key != 2] [key == 3] idx = 2; 9
[key != 1] [key != 2] [key != 3] idx = 3; -1
```

(b) After further pruning and specialization



(c) Decision tree

Fig. 3. Applying the path forest evaluation technique to Figure 2

flow can therefore be encoded as a decision tree (as in Figure 3c), and evaluating the tree to make an inference corresponds to executing the program. The path to the inferred label corresponds to the path through the program, and the label itself corresponds to the program's return value. The last step is to generate code for secure decision tree inference.

3.3 Efficient Decision Tree Inference

While we could compile down to a mux network that implements our final decision tree, we instead compile it into vectorized FHE primitives that implement the more efficient COPSE algorithm for decision tree inference [27]. The COPSE algorithm exploits the ciphertext-packing capabilities of many FHE schemes (Section 2.2) to accelerate decision tree inference by first vectorizing the computaton of each branch condition (i.e. `key == 1`, `key == 2`, etc. are all vectorized together), evaluating each level of the tree in parallel, and then accumulating the levels to obtain the final result. The COPSE algorithm is described further in Section 5.1. At this point, we can also use traditional vectorization techniques to further optimize the inference [12, 26, 33]. Section 4.5 discusses our code generation strategy in more detail.

3.4 COIL Discovers Good Implementations

COIL often automatically generates code equivalent to well-known expert-coded implementations without requiring any special programmer knowledge. For example, a well-known strategy for associative array lookup[8] involves first computing a "one-hot" indicator vector encoding the position of the key:

[8]This strategy is adapted from MPC folklore (e.g. https://www.zama.ai/post/encrypted-key-value-database-using-homomorphic-encryption) and is often used as a secure array indexing primitive in larger protocols [6, 7]

NumExpr	e	$::=$	x	variable
		$\mid$	n	numeric literal
		$\mid$	$\mathbf{ptxt}(n)$	plaintext input
		$\mid$	$\mathbf{ctxt}(n)$	ciphertext input
		$\mid$	$\mathbf{if}\ (b)\ \{e_1\}\ \mathbf{else}\ \{e_2\}$	conditional
		$\mid$	$\mathbf{let}\ x = p_1\ \mathbf{in}\ p_2$	variable definition
		$\mid$	$\mathbf{mux}(b, e_1, e_2)$	multiplexer
		$\mid$	$@arr[e_1]$	array index
		$\mid$	$\mathbf{update}\ arr\ \{\ n_1 \to e_1, \ldots, n_k \to e_k\ \}\ \mathbf{in}\ p$	array update
		$\mid$	$(e_1 \cdot e_2)$	arithmetic
		$\mid$	$f\ (e_1, \ldots, e_n)$	function call
BoolExpr	b	$::=$	$e_1 \equiv e_2$	equality
		$\mid$	$e_1 < e_2$	inequality
		$\mid$	$\neg b$	negation
CoilProgram	p	$::=$	e	numeric expression
		$\mid$	$[e_1; e_2; \ldots; e_n]$	array literal
		$\mid$	$\mathbf{ctxts}(n_1, n_2)$	ciphertext array
		$\mid$	$\mathbf{ptxts}(n_1, n_2)$	plaintext array
		$\mid$	$\mathbf{let}\ f = \lambda(x_1, \ldots, x_n) \Rightarrow \{e\}\ \mathbf{in}\ p$	function definition

Fig. 4. Syntax of the COIL language. Note that the **mux** production is only added for the purposes of the discussion in Section 7, and is not used in any of our benchmarks otherwise.

```
let idx = [key == keys[0], key == keys[1], . . . ]
```

and then computing a dot product between this vector and the values:

```
let value = dot(idx, values)
```

We can compare this strategy to the one COIL generates from the implementation in Figure 2. After pruning and unraveling all the control flow paths, the only conditionals left to compute are the ones of the form "`key == keys[i]`", which the COPSE algorithm vectorizes together, similar to the first step of the dot product strategy described above. Furthermore, COPSE's parallelized level evaluation and accumulation (described in more detail in Section 5.1) turns out to be roughly equivalent to the dot product step. Overall, given a naïve implementation of associative array lookup, COIL automatically discovers something very similar to what a cryptographic expert might write. This phenomenon is further discussed in Section 6.2.

4 DESIGN

This section first describes the COIL language (Section 4.1), then explores in more detail the transformations done in the first compilation stage (Sections 4.2-4.3), and finally discusses how COIL generates code from a path forest.

4.1 Language Design

COIL is a high-level language that supports arrays with a publicly known length, recursion, conditional expressions with both plaintext and ciphertext conditions, and basic arithmetic operators over both *encrypted* (i.e. "private" or "ciphertext") and *unencrypted* (i.e. "public" or "plaintext") inputs (Figure 4). COIL uses a *staging compiler*, which means that programs are compiled in multiple *stages* [23]. The first stage compiles a COIL program down to the *path forest IR* (Figure 5) by *partially evaluating* it with respect to the public inputs when they become available, and then performs

Expr	e	::=	n	numeric literal
		$\mid$	$\textbf{ptxt}(n)$	plaintext literal
		$\mid$	$\textbf{ctxt}(n)$	ciphertext literal
		$\mid$	$e_1 \cdot e_2$	arithmetic
		$\mid$	$\ldots$	
PathClause	c	::=	$e_1 \equiv e_2$	equality
		$\mid$	$e_1 < e_2$	inequality
		$\mid$	$\neg b$	negation
Path	p	::=	$[c_1] \ldots [c_n]$	
PathForest	f	::=	$(p_1, e_1); \ldots; (p_n, e_n)$	

Fig. 5. Syntax of the path forest IR. Note that Expr technically also includes the other syntactic forms from COIL programs, but the compilation process eventually rewrites all of these to one of the final forms listed in the grammar.

$$\llbracket (p, n) \rrbracket^{\Gamma} \triangleq (p, n) \qquad \text{where } n \text{ is a numeric, } \textbf{ptxt}\text{, or } \textbf{ctxt} \text{ literal}$$

$$\llbracket (p, x) \rrbracket^{\Gamma} \triangleq (p, \Gamma(x)) \qquad \text{if } x \text{ is a variable bound in } \Gamma$$

$$\llbracket (p_1, e_1); \ldots; (p_n, e_n) \rrbracket^{\Gamma} \triangleq \llbracket (p_1, e_1) \rrbracket^{\Gamma}; \ldots; \llbracket (p_n, e_n) \rrbracket^{\Gamma}$$

$$\llbracket (p, \textbf{let } x = e_1 \textbf{ in } e_2) \rrbracket^{\Gamma} \triangleq \llbracket (p', e_2) \rrbracket^{\Gamma[x \mapsto e']}; \ldots \qquad \text{for each } (p', e') \in \llbracket (p, e_1) \rrbracket^{\Gamma}$$

$$\llbracket (p, \textbf{if } (b)\ \{e_1\} \textbf{ else } \{e_2\}) \rrbracket^{\Gamma} \triangleq \llbracket (p'\ [b'], e_1); (p'\ [\neg b'], e_2) \rrbracket^{\Gamma}; \ldots \qquad \text{for each } (p', b') \in \llbracket (p, b) \rrbracket^{\Gamma}$$

$$\llbracket (p, f(e_1, \ldots, e_n)) \rrbracket^{\Gamma} \triangleq \llbracket (p\ p'_1 \ldots p'_n, e) \rrbracket^{\Gamma[x_i \mapsto e'_i]}; \ldots \qquad \text{where } \Gamma(f) = \lambda(x_1, \ldots, x_n) \Rightarrow \{e\}, \text{ for each } (p'_i, e'_i) \in \llbracket (p, e_i) \rrbracket^{\Gamma}$$

$$\llbracket (p, e_1 \cdot e_2) \rrbracket^{\Gamma} \triangleq \llbracket (p'_i\ p''_j, e'_i \cdot e''_j) \rrbracket^{\Gamma}; \ldots \qquad \text{for each } (p'_i, e'_i) \in \llbracket (p, e_1) \rrbracket^{\Gamma} \text{ and } (p''_j, e''_j) \in \llbracket (p, e_2) \rrbracket^{\Gamma}$$

Fig. 6. The transformations done by the COIL compiler are implemented in the $\llbracket \cdot \rrbracket^{\Gamma}$ operator, which successively rewrites terms in the path forest IR. Γ is a context mapping variables to normal-form (**let**-free, **if**-free, and function-free) expressions. The rewrite rules for **update** and array indexing are similar to the rules for **let**-bindings and arithmetic, and are omitted for clarity.

optimizations on this IR (Section 4.4) [21]. The second stage further lowers the path forest IR into vectorized FHE instructions that can execute once the ciphertext inputs are available.

To fit the encrypted computation paradigm, the language imposes restrictions on programs:

- All array indices must be publicly known[9]
- Recursion must terminate *based on publicly known inputs*, since function calls are resolved in the first stage (Section 4.3).

In the example program in Figure 2, the first restriction means that since `cur` is used as an array index in both `index` and `lookup`, it must be a plaintext input. The example satisfies the second restriction because, for instance, the recursive calls to `index` can be inlined until `cur == len(array)`, and since the condition is a plaintext, the unfolding can be done entirely in the first stage.

[9]Ciphertext indices are possible by, for example, writing an array indexing function (such as the one described in Section 3.4). The staged design of the language allows this to happen with zero abstraction overhead, as discussed later in this section.

```
let midpoint = \(x, y) => {
    if ((y - x) < 2) {
        x
    } else {
        midpoint((x + 1), (y - 1))
    }
} in
let binary_search = \(arr, key, lo, hi) => {
    let mid = midpoint(lo, hi) in ...
} in ...
```

Fig. 7. Snippet of a COIL program implementing a binary search over an array of encrypted data. While the language does not natively support division, the programmer can implement a `midpoint` function over plaintexts without incurring a run-time overhead.

4.2 Compilation

The first stage of compilation requires lowering a COIL program (Figure 4) to the path forest intermediate representation, the syntax for which is shown in Figure 5, and then optimizing the resulting forest. A term in the IR (a "forest") consists of a set of tuples (p, e), where each p can be thought of as a sequence of boolean conditions that must be true for the program to evaluate to the corresponding expression e. A tuple (p, e) is said to be in *normal form* if the expression e contains no function calls, **let**-bindings, branches, or array index/update expressions.

One way to accomplish the lowering is by embedding a COIL program $\mathcal{P}$ into the path forest IR as $(\texttt{true}, \mathcal{P})$, and then inductively applying the transformations in Figure 6 until the resulting forest is in normal form.

Of particular importance are the rewrite rules for **let**-bindings, **if**-statements, and arithmetic. Given an arithmetic expression like $e_1 \cdot e_2$, for every pair of expressions that the operands e_1 and e_2 could evaluate to, we generate a path that produces the result of applying the operation "$\cdot$" to the pair. The rule for **if** is similarly straightforward: the expression **if** (b) $\{e_1\}$ **else** $\{ e_2 \}$ can be translated into two paths; one that computes e_1 under the condition b, and one that computes e_2 under the condition $\neg b$. Finally, the rule for **let**-bindings (together with the Γ-lookup rule) implements *substitution*: when encountering an expression of the form **let** $x = e_1$ **in** e_2, we generate paths that replace x with every possible result of evaluating e_1. Note that some of the transformations described above have a multiplicative effect on the total number of paths, meaning that before optimizations, the size of the generated forest is roughly exponential in the number of ciphertext-dependent branches in the original program. In practice, however, we find that many of these branches are highly correlated, which makes some paths infeasible and thus able to be pruned (Section 4.3). This exponential effect is discussed further in Section 7.

Example. Consider the COIL expression below:

```
z + (let w = if (x < y) { y } else { x } in 2 * w)
```

Following the arithmetic and **let** rules, we first evaluate the expression **if** $(x < y)$ $\{y\}$ **else** $\{x\}$. From the **if** rule, we see that this results in a forest with two paths: "$(x < y, y); (x \geq y, x)$". Substituting these paths in for w in the **let** binding, we get: "$(x < y, 2 * y); (x \geq y, 2 * x)$". Finally, we can apply the arithmetic rule. Since the left operand (z) can only evaluate to itself, and the right operand (the

let binding) can evaluate to either $2 * y$ or $2 * x$, the final forest we get for the arithmetic expression looks like: "$(x < y, z + 2 * y); (x \geq y, z + 2 * x)$".

4.3 Optimizations on Path Forests

Recall from Section 3 that the key principle behind COIL's compilation strategy is that *path forests enable path-dependent optimization*. In this section, we describe how to adapt the rewrite rules to include the two main optimizations COIL employs: *specializing* known (plaintext) values, and *pruning* unreachable paths.

4.3.1 Specialization. Consider again the example in Section 4.2. Notice that if the variables x and y are plaintexts, then by the time we've generated the forest $(x < y, 2 * y); (x \geq y, 2 * x)$, we know the value of the expressions $2 * x$ and $2 * y$, so we can replace each expression with the result of evaluating it. This can be implemented by adding the following rule to our list:

$$[\![(p, e)]\!]^{\Gamma} \triangleq (p, \mathbf{eval}(e)) \qquad \text{if } e \text{ can be evaluated as a plaintext}$$

(Here, the **eval** function implements the usual semantics for evaluating plaintext expressions like arithmetic, etc.) Applying this optimization has a few results. Of course, this reduces the total amount of computation that needs to happen during the second (ciphertext) stage. Furthermore, since all plaintext array indices can now be computed at staging-time, the language can operate over arrays without needing the arrays to show up in the final ciphertex computation. Finally, it enables programming with *zero-cost abstractions*. For example, consider the snippet of a binary search implementation in Figure 7 in which the programmer writes a function that loops to compute the midpoint of two indices[10]. Since the `midpoint` function is only called on plaintext inputs (`lo` and `hi`), it can be executed entirely in the first (plaintext) stage, obviating the need to execute the expensive loop over ciphertext inputs in the second stage.

4.3.2 Pruning. When x and y are both plaintexts, then by the time we've generated the forest $(x < y, y); (x \geq y, x)$, we already know which of the paths is going to be taken; Hence, the other one can be removed. In particular, we replace the **if**-rule in Figure 6 with the following:

$$[\![(p, \mathbf{if}\ (b)\ \{e_1\}\ \mathbf{else}\ \{e_2\})]\!]^{\Gamma} \triangleq \begin{cases} [\![(p\ p', e_1)]\!]^{\Gamma}; \ldots & p \implies b' \\ [\![(p\ p', e_2)]\!]^{\Gamma}; \ldots & p \implies \neg b' \\ [\![(p'\ [b'], e_1); (p'\ [\neg b'], e_2)]\!]^{\Gamma}; \ldots & \text{otherwise} \end{cases}$$

When encountering a branching condition b, we generate the queries $p \wedge b$ and $p \wedge \neg b$ and discharge them to a solver[11]; if one of these queries is shown to be unsatisfiable, we avoid generating the corresponding path. In particular, this means that paths are pruned if they can be *proven unreachable based on information available at staging time*, even if the condition itself is not plaintext!

4.4 Recursive Functions

In addition to producing smaller path forests (and therefore more efficient ciphertext computation), the optimizations described above also enable COIL to gracefully compile programs with recursive functions without requiring any special care. Recall that the original form of the rewrite rules presented in Figure 6 fails to terminate in the presence of recursive functions. In particular, the original **if** rule always generates two paths, which means it *always expands the recursive case*, and

[10]Writing a function to calculate midpoints is necessary because most FHE schemes do not natively support integer division, and hence COIL does not provide a division operator.

[11]In our implementation, we discharge these queries to a lightweight custom solver capable of reasoning about linear arithmetic and inequalities. We could instead discharge to a more full-featured SMT solver and potentially be able to prune more paths at the cost of longer compile times.

hence expansion never terminates! In contrast, if at some point during compilation COIL can prove that the recursive branch does not get taken, then the new version of the **if** rule that incorporates pruning will generate *only the path for the base case*. Note that the second condition in Section 4.1 guarantees that at some point during compilation, COIL will be able to prune the recursive path.

Example. Consider the following COIL snippet which calculates the maximum value in an array of two elements:

```
let max = \(arr, cur, len, acc) => {
  if (cur == len) { acc } else {
    let newMax = if (@arr[cur] > acc) { @arr[cur] } else { acc } in
    max(arr, cur + 1, len, newMax)
  }
} in max(arr, 0, 2, 0)
```

Applying the function call rule gives us:

```
if (0 == 2) { 0 } else {
  let newMax = if (@arr[0] > 0) { @arr[0] } else { 0 } in
  max(arr, 1, 2, newMax)
}
```

Now, we apply the *new* **if** rule, which immediately prunes the first branch of the conditional and yields:

```
let newMax = if (@arr[0] > 0) { @arr[0] } else { 0 } in
max(arr, 1, 2, newMax)
```

and then the forest:

$$(arr[0] > 0, \mathtt{max}(arr, 1, 2, arr[0])); (arr[0] \leq 0, \mathtt{max}(arr, 1, 2, 0))$$

Expanding the recursive call in each path again yields the forest:

$$\begin{aligned}
&(arr[0] > 0 \wedge arr[1] > arr[0], && \mathtt{max}(arr, 2, 2, arr[1]));\\
&(arr[0] > 0 \wedge arr[1] \leq arr[0], && \mathtt{max}(arr, 2, 2, arr[0]));\\
&(arr[0] \leq 0 \wedge arr[1] > 0, && \mathtt{max}(arr, 2, 2, arr[1]));\\
&(arr[0] \leq 0 \wedge arr[1] \leq 0, && \mathtt{max}(arr, 2, 2, 0))
\end{aligned}$$

Finally, for each call to `max(arr, 2, 2, ...)` COIL can immediately prove `2 == 2` and thus expand only the base case, yielding the forest:

$$\begin{aligned}
&(arr[0] > 0 \wedge arr[1] > arr[0], && arr[1]);\\
&(arr[0] > 0 \wedge arr[1] \leq arr[0], && arr[0]);\\
&(arr[0] \leq 0 \wedge arr[1] > 0, && arr[1]);\\
&(arr[0] \leq 0 \wedge arr[1] \leq 0, && 0)
\end{aligned}$$

which computes the maximum as desired.

4.5 Generating Code

The second stage of the COIL compiler takes the optimized path forest produced from the steps described above, and uses it to generate FHE code that operates on ciphertext inputs. The basic code generation strategy starts by using the COPSE algorithm [27] to generate code that simultaneously executes each path in the forest, vectorizing together all the path conditions at each step to produce a ciphertext that encodes which path has all its conditions satisfied. The expressions at the end of each path are then vectorized together, and the ciphertext is used to select the correct expression to return. Section 5.1 describes the details of the COPSE algorithm.

5 IMPLEMENTATION

In this section, we discuss the implementation of COIL: the specifics of the vectorized decision tree inference algorithm (Section 5.1) and our choice of FHE scheme (Section 5.2).

5.1 Efficient Decision Tree Evaluation via COPSE

COPSE is an algorithm for performing private decision tree inference that supports multithreading and takes advantage of the vectorizing capabilities of RLWE-based encryption schemes [27] (Section 2.2). The COPSE algorithm consists of three steps:

1. *Comparison:* All the branching conditions in the tree are vectorized together and evaluated simultaneously
2. *Level Processing:* For each depth level of the tree, all the branches at that level are analyzed to exclude unreachable labels. This is done via vectorized matrix operations, and each level can be processed in parallel.
3. *Accumulation:* The results from processing each level are combined to determine the single label representing the result of evaluating the tree

The COIL implementation slightly modifies the *Comparison* step described above. The original COPSE algorithm evaluates decision trees in which the branching conditions are all of the form $x_i < \alpha_i$. COIL relaxes this assumption to allow for decisions of the form `exp1 < exp2` or `exp1 == exp2` for arbitrary expressions `exp1` and `exp2`, such as the tree in Figure 8. In particular, we first vectorize together all the `exp1` and (separately) all the `exp2` (e.g. computing vectors `[a; b; a-b]` and `[b; a; b]`), then compare the resulting vectors using *both* == *and* <, and finally blend the comparison results together to correspond to the actual sequence of decisions in the tree. For the expression vectorization we use Coyote, an off-the-shelf vectorizer specifically designed for FHE applications [26].

5.2 Choice of FHE Scheme

Here we justify our choice of FHE scheme for the COIL backend; namely, the mod-2 variant of the BFV/BGV scheme.

FHE schemes can be broadly characterized along two dimensions: *vectorized* vs *unvectorized* schemes, and *boolean* vs *arithmetic* schemes. Vectorized schemes, such as CKKS and BFV/BGV, support *ciphertext packing* (Section 2.2), which allows for computing multiple operations in parallel at the cost of each operation being relatively slow. In contrast, individual homomorphic operations in unvectorized schemes such as TFHE and CGGI tend to be much faster, but these schemes can only execute one operation at a time. In order to fully take advantage of the COPSE algorithm's vectorizability, we restrict ourselves to vectorized schemes.

In boolean schemes like TFHE, CGGI, and mod-2 variants of BFV/BGV, ciphertexts are *encryptions of bits*, and primitive homomorphic operations correspond to logical gates like AND and XOR. In arithmetic schemes like CKKS and mod-p variants of BFV/BGV, ciphertexts instead encrypt

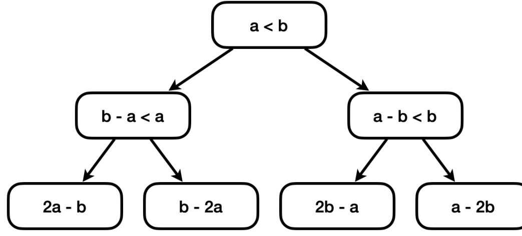


Fig. 8. Decision tree computing a bounded GCD

integers[12], with the primitive homomorphic operations being addition and multiplication. While arithmetic operations are much more efficient in arithmetic schemes (evaluating a single addition operation is much cheaper than evaluating a binary adder circuit), they struggle with computing non-smooth functions that are not easily approximated by a polynomial. Since COIL primarily targets computations with branching decisions, and the comparison function $f(x, y) = x < y$ is non-smooth, our backend needs to use a boolean scheme, and this in particular restricts us to using mod-2 BFV/BGV.

6 EVALUATION

To determine the effectiveness of the compilation techniques presented in this paper, we ask the following research questions:

- **RQ1: How efficient are the programs COIL generates compared to other compilation strategies?** We compile a set of benchmarks with COIL and compare the run times against those of naive implementations[13]
- **RQ2: How does COIL compare to known custom protocols?** We compare the COIL run times to those of expert-designed protocols for a subset of our benchmarks
- **RQ3: What kinds of optimizations does COIL enable?** We discuss the challenges associated with implementing our `merge` benchmark, and analyze how COIL's unique compilation strategy enables optimization opportunities that address these challenges.

All experiments are run on 2020 M1 MacBook Air with 16GB of RAM; the values reported are the medians across eleven runs and a 95% confidence interval.

6.1 How efficient are the programs COIL generates?

There is no standard set of FHE benchmarks, and especially no set that make use of conditionals over ciphertext. We evaluate COIL on the following set of benchmarks that rely on conditionals and implement several common kernels:

1. `linear_index`, looking up a *ciphertext* index into an array of 16 elements via a linear scan
2. `log_index`, looking up a *ciphertext* index into an array of 16 elements via a binary search
3. `sp_auction`, determining the winning bidder and bid in a second-price auction with 8 bidders
4. `filter`, using a threshold predicate to filter a list of 8 elements
5. `merge`, merging two sorted 5-element arrays into a sorted array of 10 elements
6. `associative_array`, using a secure key to look up a value in an associative array of 8 elements

[12]Technically, CKKS ciphertexts encrypt rational numbers with respect to some fixed-point precision.
[13]Our naive implementations are manually transliterated from the COIL surface language to C++. An example of this translation is shown in Figure 10b.

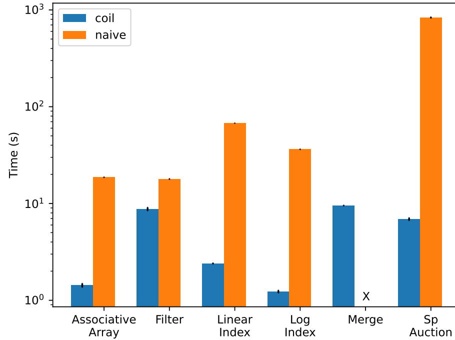


Fig. 9. Running time of each benchmark. The reported COIL times include both the online and offline phase. Naïve `merge` times out after 30 minutes.

Benchmark	Stage 1	Stage 2	Total
`linear_index`	25 ms	290 ms	315 ms
`log_index`	48 ms	263 ms	311 ms
`sp_auction`	57 ms	370 ms	427 ms
`merge`	120 ms	6730 ms	6850 ms
`filter`	31 ms	1539 ms	1570 ms
`associative_array`	24 ms	277 ms	301 ms

Table 1. COIL compile times broken down by stage

Each benchmark is implemented in COIL's surface language (Figure 4), and compiled down to calls to the HElib library [20] using the mod-2 BGV scheme.

The compilation times for each benchmark are reported in Table 1, broken down by stage. Notice that with the exception of `merge`, the compilation time is negligible (on the order of a few hundred milliseconds). The high compile time for merge can be attributed to the combinatorial explosion of paths, a phenomenon which is discussed in more detail in Section 6.3.

The time each benchmark takes to run is reported in Figure 9. We see that COIL outperforms naïve implementations, sometimes by as much as two orders of magnitude. These speedups are most prominent on the benchmarks containing instances of an array being indexed by a ciphertext, as COIL's unique specialization and pruning strategy is able to avoid the overhead of these expensive indexing operations. In fact, the data for the naïve implementation of the `merge` benchmark is missing: even on modest input sizes (e.g. merging two arrays of size 5) it times out after 30 minutes. Section 6.3 analyzes where COIL's speedups come from on this benchmark.

6.2 How does COIL compare to known custom protocols?

With some of our benchmarks, a naïve translation of a COIL implementation is unfair, as we already know of efficient specialized protocols that implement them. In particular, we can use the one-hot vector + dot product strategy described in Section 3.4 for `linear_index`, `log_index`, and `associative_array`, a single vectorized comparison + mux for `filter`, and part of a custom k-way sorting protocol for `merge` [22].

Benchmark	COIL time (s)	Expert time (s)
linear_index	2.4	11.4
log_index	1.2	11.4
associative_array	1.4	12.5
filter	8.8	1.9
merge	9.5	84.2

Table 2. Comparing COIL to custom implementations

```
let merge =
  \(arr1, arr2, out, i1, i2, j) => {
    ...
    if (@arr1[i1] < @arr2[i2]) {
      update out {j := @arr1[i1]} in
      merge(arr1, arr2, (i1 + 1), i2, out j)
    } else {
      update out {j := @arr2[i2]} in
      merge(arr1, arr2, i1, (i2 + 1), out, j)
    }
    ...
  } in ...
```

(a) Implementation of merge in COIL

```
ctxt i1, i2 = encrypt(0);
for (int i = 0; i < len(arr1) + len(arr2);
     i++)
{
  ctxt arr1Val = index(arr1, i1);
  ctxt arr2Val = index(arr2, i2);
  ctxt b = compare(arr1Val, arr2Val);
  output[i++] = mux(b, arr1Val, arr2Val);
  i1 = mux(b, i1 + 1, i1);
  i2 = mux(b, i2, i2 + 1);
}
```

(b) Naïve implementation of merge in C++

Fig. 10. Example snippets merging two sorted arrays of ciphertexts in C++ and in the COIL surface language. Note that an actual implementation of `merge` would include bounds checks for the two indices `i1` and `i2`; these checks have been omitted from the above code for the sake of clarity.

Table 2 compares the COIL's running time to each of these custom protocols. The results are summarized in Notice that with the exception of `filter`, the COIL version *outperforms* its associated custom protocol! The poor performance of COIL on `filter` is perhaps unsurprising: a majority of COIL's benefits come from its ability to specialize ciphertext computation into plaintext, and this is a pattern largely missing from the `filter` benchmark. In fact, even the naïve implementation does reasonably well (though notably, not as well as COIL!) just by looping over the array, performing a comparison and conditional write at each (plaintext) index.

In contrast, we see significant speedups on other benchmarks. This is, again, unsurprising for secure array indexing and associative array lookup: Recall from Section 3.4 that the COIL evaluation strategy essentially recovers something very similar to the usual dot product strategy, and the fact that we can parallelize decision tree evaluation further speeds up the dot products [27]. Explaining the speedup for `merge` is far more interesting, and its worth taking some time to dissect this benchmark properly.

6.3 Where do COIL's speedups come from?

Figure 10 shows two partial implementations of a function that merges sorted arrays, one in COIL's surface language and one that naïvely translates it to C++. We first analyze the naïve implementation, and then walk through what COIL's evaluation strategy does to improve it.

What does a naïve merge look like? Consider the snippet in Figure 10b which naïvely implements a `merge` function in C++. Each iteration of the for loop contains:

(1) Two secure array index operations

(2) A comparison on the results of the lookups
(3) Two oblivious updates to the array indices

The naïve implementation performs *multiple* inefficient secure array lookups, each of which contributes nontrivially to the overall multiplicative depth. Furthermore, the entire procedure is inherently sequential, making it difficult to recover performance via parallelization or vectorization. Finally, since each iteration uses the ciphertext index values updated from the previous iteration, the total multiplicative depth *stacks*, resulting in circuits that either require huge parameters to evaluate or multiple expensive rounds of bootstrapping. Given this, it is perhaps unsurprising that our naïve implementation of `merge` times out after running for 30 minutes (Figure 9).

What does COIL do differently? The COIL evaluation strategy transforms the snippet in Figure 10a into something much more efficient. Each specialization/pruning step inlines one level of the recursive `merge` call, eventually unfolding the entire program into a large binary tree representing *every possible way to merge the lists*. Since every node in the binary tree corresponds to a single possible pair of values for `i1` and `i2`, each of the array indexing expressions (`@arr1[i1]` and `@arr2[i2]`) are specialized to those particular indices, eliminating the need for secure indexing. Finally, the decision tree protocol chooses which merged array to select.

First, by specializing all the array indices into plaintext values, we obviate the need for the expensive secure index operations that the naïve implementation uses; in particular, this greatly reduces the multiplicative depth of the overall circuit, allowing it to be evaluated with smaller parameters (or without as much bootstrapping). Second, by separating out all the paths through the function we greatly increase the amount of parallelism available: every array comparison can be evaluated in parallel, and selecting the correct path at the end can be done via efficient parallelized and vectorized operations [27]. In fact, notice that these exactly solve the issues with the naïve implementation!

At this point the reader may notice that there are an exponential number of ways to merge two lists, and hence an exponential number of paths to evaluate. Certainly, separating out all the control flow paths *seems* like a very bad idea: the amount of work increases from $O(n^2)$ to something like $O(4^n)$. However, we also go from having to evaluate everything sequentially to having multiple degrees of parallelism, as well as being able to exploit ciphertext batching. Of course, at some point the size of the arrays grows beyond what the (admittedly large) FHE vectors can reasonably hold, and we need a new approach. Indeed, there are several optimizations that can be performed on top of the version of the COIL strategy presented here to allow it to scale to significantly larger programs. These are discussed in more detail in the next section.

7 DISCUSSION

In this section we briefly describe possible ways to deal with the exponential number of paths that can result from applying the path forest strategy to certain programs.

7.1 Control Flow Linearization

Recall from the discussion in Section 6.2 that the `filter` benchmark benefits very little from path forest strategy compared to the optimized protocol, in particular because the benchmark does not contain many instances of ciphertext computation that can be specialized to plaintext. In other words, COIL still incurs the overhead of unfolding every path without being able to make up for it. One might instead consider a strategy that avoids unfolding branches which do not result in a benefit from pruning and specialization, and hence are not “worth it”. This, in fact, corresponds to a well-known technique called *control flow linearization*, in which branches are eliminated by turning a control flow dependence into a data dependence (such as a mux). [1, 3, 15, 24, 28]

Table 3. Effect of manually linearizing `filter` benchmark

`filter.lin`	`filter.naive`	`filter.coil`	`filter.expert`
5.3s	17.9s	8.9s	1.88s

Control flow linearization is entirely compatible with COIL's compilation strategy. A `mux` function is, from the perspective of the language, just another branchless computation: By manually rewriting certain branches as muxes, the programmer can ensure that they do not get unfolded by the path forest conversion process, and thus do not contribute to the exponential blowup. We investigate the effects of manual linearization by adding a ternary muxing operator to the COIL language and rewriting the `filter` benchmark to use this instead of explicit branches. Since Coyote does not currently support vectorizing ternary operations, we compile "`mux(b, x, y)`" into the arithmetic expression "`b * (x - y) + y`". The results of this manual linearization are shown in Table 3. As expected, the manually linearized version of `filter` outperforms both the naive and the branching COIL implementations, although, notably, the expert implementation is still better since it uses an efficient `mux` instruction instead of simulating it with arithmetic. However, these preliminary results demonstrate that the techniques presented in this paper can be used in conjunction with traditional control flow linearization techniques to mitigate the path explosion problem.

Of course, for programs more complex than `filter`, writing out the full multiplexed circuit by hand is tedious and sometimes prohibitively difficult. A more sophisticated compiler can perform an analysis to determine which branches should not get unfolded, and in a compilation pass before COIL automatically replace these branches with muxes; applying the COIL strategy presented in this paper to the resulting program will correctly linearize the identified branches and unfold the others. While our current implementation of COIL does not perform this analysis, it is an interesting future direction to pursue.

7.2 Blocking

While COIL can amortize much of the exponential blowup via vectorization and parallelism, FHE vector widths are not infinite. This restriction is particularly relevant to the COPSE algorithm, which treats decision tree evaluation as a series of matrix multiplications, and exploits ciphertext batching by packing matrices into ciphertext vectors large enough to hold them. A common workaround is *blocking*: If a particular set of FHE parameters allows for vectors capable of operating on 1000×1000 matrices, we can operate on an 8000×8000 matrix simply by blocking it into 64 submatrices and then operating on each submatrix. Of course, blocking gives up the asymptotic benefits of vectorization as the demand for vector lanes increases. However, FHE vectors are incredibly wide, still allowing for significant speedups over the alternative.

8 RELATED WORK

8.1 Compiling Oblivious Control Flow

The Google Transpiler [18] compiles a subset of C++ into FHE calls. The Transpiler is notably different from the other compilers listed above in that it can *handle oblivious control flow*: it uses a boolean circuit-based backend, so non-polynomial comparison operators are straightforward to implement, and conditionals can be emulated via muxes. The particular FHE backend that the Google Transpiler uses is TFHE [11], a binary-only scheme that is not based on the Ring Learning with Errors (RLWE) problem. While TFHE allows for incredibly fast bootstrapping and hence lends itself well to efficient implementations of large binary circuits, it does *not* support the

ciphertext batching optimization that RLWE schemes support (Section 2.2), making it unsuitable for vectorization.

It's worth pointing out that circuit-vectorizing compilers like Coyote [26] *can* be extended to support oblivious control flow. Coyote makes no assumptions about the plaintext modulus over which the circuit operates, so conditional statements can be compiled into muxes as usual and the resulting boolean circuit can be given to Coyote to vectorize. However, in practice the generated boolean circuits are too large and unwieldly to be able to effectively vectorize.

8.2 Reducing Control Flow

The problem of control flow is not unique to FHE. For programs running on GPUs, branching control flow can mean some threads sit idle for parts of the program, resulting in low utilization and poor performance. Various techniques have been proposed to deal with this problem [1, 3, 15, 24, 28–30, 32]. Many of these techniques often either rely on instruction set features missing from most FHE backends, or end up doing something semantically equivalent to muxes anyway. However, as we noted in Section 7, applying some control flow reduction techniques to programs before using COIL can help further improve performance.

8.3 Compilers and Optimizations for FHE

A number of FHE compilers [4, 9, 12–14, 18, 26, 33] provide a high-level surface language to write programs in, and then compile these programs down into circuits that can be executed within FHE. While most of these compilers do some optimizations on the generated circuits, these optimizations do not always include vectorization. EVA [13] *does* support packed arithmetic, but requires the programmer to do the vectorization manually. The FHE compilers that do automatic vectorization [12, 26, 33] fall into two broad camps: those that attempt to vectorize arbitrary code at the circuit level [12, 26], and those that restrict the surface language to make programs more vectorizable by construction [14, 33]. The compilers in the first camp, Porcupine [12] and Coyote [26], use expensive search procedures and synthesis techniques to automatically lift arbitrary arithmetic circuits into ones that operate over packed data.

In the other camp, Airduct [14] is an array-based intermediate language for writing multiparty computation programs. By requiring the programmer to express their computations using high-level array operations, Airduct ensures that programs are naturally amenable to vectorization. HECO [33] is an FHE language and compiler that takes advantage of arrays and array indexing in the surface language to aid in making vectorization decisions. Working at this higher level rather than at the level of circuits allows HECO to vectorize tensor programs more effectively, but the language restrictions in both HECO and Airduct preclude support for the kinds of oblivious control flow handled by COIL.

9 CONCLUSION

This paper presents COIL, a language and evaluation strategy for FHE programming with secure control flow. We show how COIL can transform programs with secure control flow into path forests, and present a strategy that executes these path forests via efficient decision tree evaluation protocols. We demonstrate on a variety of benchmarks that COIL outperforms other state-of-the-art techniques by up to an order of magnitude. COIL thus represents the next major step towards making FHE programs more efficient.

REFERENCES

[1] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of Control Dependence to Data Dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*

(Austin, Texas) *(POPL '83)*. Association for Computing Machinery, New York, NY, USA, 177–189. https://doi.org/10.1145/567067.567085

[2] Asma Aloufi, Peizhao Hu, Harry W. H. Wong, and Sherman S. M. Chow. 2019. Blindfolded Evaluation of Random Forests with Multi-Key Homomorphic Encryption. Cryptology ePrint Archive, Paper 2019/819. https://eprint.iacr.org/2019/819 https://eprint.iacr.org/2019/819.

[3] 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.

[4] 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

[5] Gilad Asharov, Abhishek Jain, and Daniel Wichs. 2011. Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE. Cryptology ePrint Archive, Report 2011/613.

[6] 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

[7] 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.

[8] Zvika Brakerski, Craig Gentry, and Shai Halevi. 2012. Packed Ciphertexts in LWE-based Homomorphic Encryption.

[9] 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

[10] Long Chen, Zhenfeng Zhang, and Xueqing Wang. 2017. Batched Multi-hop Multi-key FHE from Ring-LWE with Compact Ciphertext Extension. In *TCC*.

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

[12] 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

[13] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 546–561. https://doi.org/10.1145/3385412.3386023

[14] Vivian Ding, Coşku Acay, and Andrew C. Myers. 2023. An Array Intermediate Language for Mixed Cryptography. In *FCS*.

[15] Jeanne Ferrante and Mary Mace. 1985. On Linearizing Parallel Code. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '85)*. Association for Computing Machinery, New York, NY, USA, 179–190. https://doi.org/10.1145/318593.318636

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

[17] 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.

[18] 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.

[19] 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.

[20] Shai Halevi and Victor Shoup. 2012. Design and Implementation of a Homomorphic-Encryption Library.

[21] Christoph A. Herrmann and Tobias Langhammer. 2006. Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. *Science of Computer Programming* 62, 1 (2006), 47–65. https://doi.org/10.1016/j.scico.2006.02.002 Special Issue on the First MetaOCaml Workshop 2004.

[22] Seungwan Hong, Seunghong Kim, Jiheon Choi, Younho Lee, and Jung Hee Cheon. 2021. Efficient Sorting of Homomorphic Encrypted Data with k-way Sorting Network. Cryptology ePrint Archive, Paper 2021/551. https://doi.org/10.1109/TIFS.2021.3106167 https://eprint.iacr.org/2021/551.

[23] Ulrik Jørring and William L. Scherlis. 1986. Compilers and Staging Transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida) *(POPL '86)*. Association for Computing Machinery, New York, NY, USA, 86–96. https://doi.org/10.1145/512644.512652

[24] Yunsup Lee, Ronny Krashinsky, Vinod Grover, Stephen W. Keckler, and Krste Asanović. 2013. Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–11. https://doi.org/10.1109/CGO.2013.6494995

[25] N. Li, T. Zhou, X. Yang, Y. Han, W. Liu, and G. Tu. 2019. Efficient Multi-Key FHE With Short Extended Ciphertexts and Directed Decryption Protocol. *IEEE Access* 7 (2019), 56724–56732.

[26] 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

[27] Raghav Malik, Vidush Singhal, Benjamin Gottfried, and Milind Kulkarni. 2021. Vectorized Secure Evaluation of Decision Forests. 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, 1049–1063. https://doi.org/10.1145/3453483.3454094

[28] 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

[29] Simon Moll and Sebastian Hack. 2018. Partial Control-Flow Linearization. *SIGPLAN Not.* 53, 4 (jun 2018), 543–556. https://doi.org/10.1145/3296979.3192413

[30] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. 2022. DARM: Control-Flow Melding for SIMT Thread Divergence Reduction. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, 28–40. https://doi.org/10.1109/CGO53902.2022.9741285

[31] Nigel Smart and Frederik Vercauteren. 2011. Fully homomorphic SIMD operations. *IACR Cryptology ePrint Archive* 2011 (01 2011), 133. https://doi.org/10.1007/s10623-012-9720-4

[32] Ryan Taylor and Xiaoming Li. 2011. Software-Based Branch Predication for AMD GPUs. *SIGARCH Comput. Archit. News* 38, 4 (jan 2011), 66–72. https://doi.org/10.1145/1926367.1926379

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