COMPILER CRYPTOSYSTEM CO-DESIGN

by

Raghav Malik

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Elmore Family School of Electrical and Computer Engineering

West Lafayette, Indiana

December 2025

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Milind Kulkarni, Chair

School of Electrical and Computer Engineering

Dr. Benjamin Delaware

School of Computer Science

Dr. Tiark Rompf

School of Computer Science

Dr. Xiaokang Qiu

School of Electrical and Computer Engineering

Approved by:

Dr. Milind Kulkarni

Cancel me not - for what then shall remain?
Abscissas some mantissas, modules, modes,
A root or two, a torus and a node:
The inverse of my verse, a null domain.

Ellipse of bliss, converge, O lips divine!
The product of our scalars is defined!
Cyberiad draws nigh, and the skew mind
Cuts capers like a happy haversine.

I see the eigenvalue in thine eye,
I hear the tender tensor in thy sigh.
Bernoulli would have been content to die,
Had he but known such $a^2 \cos 2\varphi$

From "Love and Tensor Algebra," Stanislaw Lem

ACKNOWLEDGMENTS

It is, I think, traditional to start the acknowledgments of a dissertation with some pithy phrases that point out how difficult a PhD is to complete alone, how vital it is to have a good support system, and so on. While I'm certain I knew all these facts *academically* before starting my degree, its only now, as I look back over the past six years, that I realize how much this was *truly a group effort*—I don't think I could have been even a tenth as successful if I hadn't been lucky enough to be surrounded by a loving and supportive family, a wonderful group of friends and colleagues, and an incredible advisor. The degree may have my name on it, but it is as much their accomplishment as mine.

The actual list of people I'd like to thank here is far longer than what the space allows; what follows is my best approximation thereof. First and foremost, I'd like to thank my family, who have been immensely supportive throughout this endeavor. From my parents, who have been there at every turn to celebrate my successes and soften the blow of my many failures, to my little brother, who insists on accepting credit for reading the many drafts I've sent his way, to my grandparents, who I am certain are even more excited than I am at my finally graduating, I truly could not have done this without them.

I'd like to thank my D&D groups over the years for injecting some much-needed whimsy into my life: Morana, Guthard, Trym, Margaret, Ciru, Wern, Claire, Aster, Willow, Lark, Hickory, Fig, Rosalia, and many others; I won't soon forget the stories we made.

I'd like to thank Amy, Adam, and all my orchestra stand partners and chamber groupmates, including, Riley, Ben, Janhavi, Andrew, Kevin, Sreesha, Elle, Maggie, and Allison, for allowing me to continue enriching my life with music and for providing welcome distractions and outlets when research became too stressful.

I'd like to thank my labmates, Vani, Durga, Pratyush, Aditha, Krish, Charitha, Dulani, Vickrant, Artem, Vidush, Fouad, and Vedant, for helping create a welcoming and exciting work environment. I will miss the hours spent in front of whiteboards, the group dinners and discussions, and the countless post-deadline nights at Knickerbocker and the Spot.

I'd like to thank Anja, David, Tiana, Pranesh, Ilan, Abhi, and Jenna, for helping me discover a love of teaching, and for listening to my innumerable rants about C++.

I'd like to thank Omar, Kurt, Sonya, Camille, Sunidhi, Jenna, Sara, Anja, Pratyush, Patrick, Dharun, Aditi, Alex, Juli, Nicole, and Jacob, for being amazing friends, for putting up with me for so long and for allowing me to have some semblance of a social life.

I'd like to thank Donnie for reminding me about Stirling's Approximation at every opportunity.

I'd particularly like to thank Sofia, Manuel, Francille, Ben, and Qianchuan, for engaging with and encouraging my slow descent into category-theoretic madness, despite many others' best efforts to the contrary.

I'd like to thank the members of my committee, Ben, Tiark, and Xiaokang, for their feedback and advice over the years; this document would be far less polished without it.

Finally, I'd like to thank Milind. I am convinced that one's PhD advisor contributes much more to one's experience than the topic of one's dissertation, and I don't think I could have asked for a better advisor than Milind. When I started my degree I had no idea I wanted to do PL, or even what PL really entailed; I just knew the project he had in mind sounded interesting, and I wanted to work with the professor who taught me C all those years ago. Over the years, he has infected me with his enthusiasm for the subject and for academia, helped me develop my own research ideas and writing style, roasted me (probably less than I deserve) for constantly bringing up category theory, and molded me into the researcher I am today. I am grateful for the chance to have been one of his students.

This work was supported by the National Science Foundation grants CCF-2216987, CCF-1919197, CCF-1908504, CCF-1725672, the Office of the Director of National Intelligence, Intelligence Advanced Research Projects Activity, contract #2019-19020700004, and Cisco.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

BFV	Brakerski-Fan-Vercauteren
CGGI	Chillotti-Gama-Georgieva-Izbachène
DSL	Domain-Specific Language
FHE	Fully Homomorphic Encryption
GDPR	General Data Protection Regulation
HEIR	Homomorphic Encryption Intermediate Representation
HIPAA	Health Insurance Portablity and Accountability Act
IR	Intermediate Representation
ISA	Instruction Set Architecture
LUT	Lookup Table
MLIR	Multilevel Intermediate Representation
Mux	Multiplexer
MPC	Multiparty Computation
PBS	Programmable Bootstrapping
RLWE	Ring Learning With Errors
SIMD	Single Instruction Multiple Data
SLP	Superword-Level Parallelism

GLOSSARY

Alignment	The part of a vector schedule that describes which instructions (gates) are packed together
Arithmetic Circuit	A circuit in which the wires hold integers and the gates represent integer addition/multiplication
Arithmetic Lookup Table	A multi-input lookup table in which the inputs are interpreted as an index into the table via a linear combination
Boolean Circuit	A circuit in which the wires hold bits and the gates represent Boolean gates
Bootstrapping	An operation in Fully Homomorphic Encryption schemes that "refreshes" a ciphertext, allowing computation over that ciphertext to continue
Control Flow	The order in which program statements are executed
Ciphertext	Describing an object that "contains" a (plaintext) value, but is indistinguishable from a random value without the secret key
Ciphertext Packing	The ability of certain (RLWE-based) FHE schemes to pack multiple plaintexts into a single ciphertext, and perform computations element-wise
Circuit	A directed acyclic graph in which vertices are *gates*, or simple functions being evaluated, and edges are *wires* dataflow dependences between the gates.
Cryptosystem	A set of algorithms for encrypting plaintexts and decrypting ciphertexts
Data Movement	The instructions in a vector program that move data between lanes to align them for future vector operations
Decision Tree	A program with branching control-flow, and a unique control-flow path to each basic block

Fully Homomorphic Encryption	Any cryptosystem that additionally provides the capability to evaluate arbitrary finite circuits directly over ciphertexts
Lane Placement	The part of a vector schedule that describes the vector lane on which each instruction (gate) produces its result
Lookup Table	A unary function with domain a finite totally ordered set such as $\mathbb{Z}/p$ called the set of *rows*
Multiparty Computation	Any cryptographic protocol that allows mutually distrustful parties to collaboratively compute a known function over their private inputs
Multiplexing	A technique for linearizing branching control flow by evaluating both branches and then selecting a single result
Path Forest	A data structure that annotates control flow paths through a program with path-dependent information
Plaintext	Describing a value that has yet to be encrypted, and does not require a secret key to interpret
Programmable Bootstrapping	The ability of CGGI to evaluate arbitrary (unary, negacyclic) functions while bootstrapping
Protoschedule	A vector schedule for a quotient of a circuit
Quotient	(of a graph) A graph related to the original by contracting the edges of a connected subgraph
Vector Lane	A slot in a vector that can hold a single unit of data
Vector Schedule	For a circuit, a choice of which gates get executed simultaneously, and an assignment of a vector lane on which the output of each gate is produced
Vectorization	A compiler pass that transfoms a program expressed over individual data elements into one expressed over *packed vectors of data*

ABSTRACT

In a world increasingly concerned with data privacy, the availability of privacy-preserving computation is more important than ever. Unfortunately, the widespread adoption of powerful cryptographic techniques such as Multiparty Computation (MPC) and Fully Homomorphic Encryption (FHE) is limited by the great deal of expertise they require, and their lack of easy programmability. Alleviating this burden of expertise is the subject of much research from both the programming languages and cryptography communities, and is the goal of this dissertation. To this end, we introduce the philosophy of *Compiler Cryptosystem Co-Design*, which posits that in order to achieve this goal, we need to *build compilers that understand the target cryptosystem*, and simultaneously *develop cryptosystem abstractions tailored to the compiler* rather than the programmer.

We justify the above claim by presenting four increasingly ambitious examples of applying Compiler Cryptosystem Co-Design to problems in the domain of Fully Homomorphic Encryption computations: Coyote, COPSE, COIL, and COATL. In Coyote, we build a compiler with enough context to understand the subtleties of FHE vectorization, and demonstrate how this allows it to outperform traditional vectorization techniques. With COPSE we explore the other side of this philosophy via a highly vectorizable cryptographic abstraction that encodes the branching structure of (encrypted) decision trees; We show how to use this abstraction to accelerate an otherwise difficult-to-parallelize problem like secure decision forest inference. In COIL we push the idea of "co-design" to its natural conclusion by combining the insights from the previous two examples in a language for expressing secure computations that contain nontrivial control flow. Finally, with COATL we go a step further, *changing the underlying programming model of the cryptosystem* while developing compilation techniques to perform optimizations that were previously inexpressible.

We conclude by observing that although this dissertation represents an important step towards making privacy preserving computation universally accessible, the ideas contained therein barely scratch the surface of what is possible with compiler cryptosystem co-design; we present two interesting future directions to explore.

1. INTRODUCTION

Whether it is to protect intellectual property, as in the case of secure machine learning, or to comply with governmental data privacy regulations like HIPAA or GDPR, or simply to defend against unwanted data breaches, we often need to write *privacy-preserving applications* that can perform computations without learning anything about their inputs.

A popular approach to writing privacy-preserving software is Multiparty Computation (MPC), a class of cryptographic protocols that allow mutually distrusting parties to jointly compute functions over private inputs without sharing those inputs with each other. While MPC protocols are, in theory, powerful enough to evaluate any (bounded, terminating) function, actually writing practical programs that use MPC has a few drawbacks:

- The high cryptographic overhead of MPC sometimes makes nontrivial programs prohibitively expensive to evaluate
- The MPC programming model can be highly counterintuitive, making it very difficult for non-experts to write efficient MPC programs

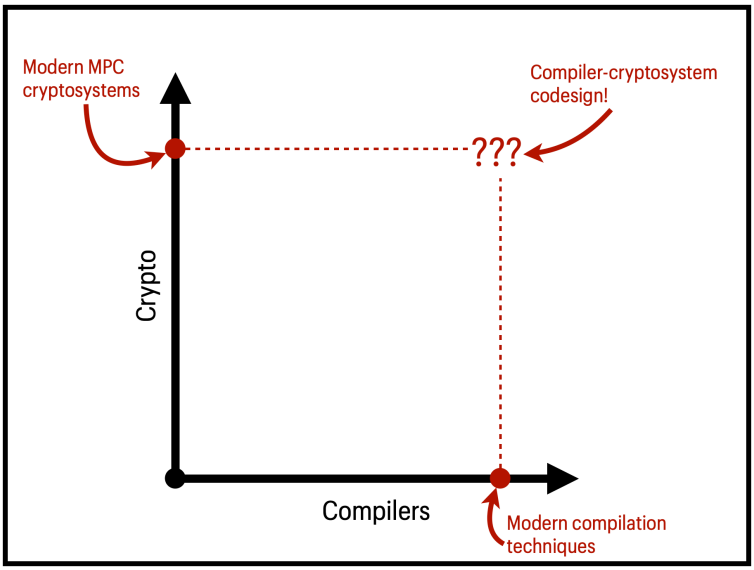


Figure 1.1. Compiler Cryptosystem Co-Design can bridge the gap between sophisticated but crypto-agnostic compiler analyses, and modern cryptosystems with rich capabilities.

MPC-Aware Compilers

Compilers naturally suggest themselves as a solution to both these problems: By treating the low-level MPC primitives as a fixed instruction set to target, we can allow programmers to express their computations in a high level language, and automatically lower them down to our "MPC instruction set". Of course, this is by no means a novel insight; the literature is littered with examples of MPC languages and compilers that mechanically translate insecure operations into their secure MPC equivalents, but do very little optimization on top of this [1–10]. This is essentially how traditional compilers worked in the 1980s—translating high-level languages into assembly one line at a time—but compilation techniques have come a long way since then, and optimizations like smart instruction selection and vectorization are practically standard in any modern production-quality compiler (as illustrated on the horizontal axis of Figure 1.1).

Unfortunately, applying techniques like these to the MPC space can be highly nontrivial. Consider, for example, Superword-Level Parallelism (SLP), a technique for automatically vectorizing programs even in the absence of structures like for-loops [11–13]. Vectorizing compilers that use SLP typically target vector instruction sets in CPUs, and SLP therefore bakes in certain assumptions about the cost model of these instructions. A naive attempt at applying SLP-style vectorization to MPC programs fails for essentially this reason: While some MPC schemes do support a limited form of vectorization called *ciphertext packing*, their programming model is vastly different from that of CPU vector architectures. In particular, the lack of easy vector shuffle/select instructions means that SLP-vectorized code can easily perform *worse* than its scalar counterpart—a phenomenon discussed in more detail in Chapter 3, in which we develop techniques to adapt SLP-style vectorization to the MPC domain. The takeaway is that just as a good compiler engineer needs to have a deep understanding of the target architecture, *writing a good MPC compiler requires a deep understanding of the semantics and cost model of the target MPC scheme!*

Compiler-Aware MPC

The story so far is reasonably satisfying: We can build compilers with a "baked in" understanding of MPC-specific optimizations, turn these loose on our privacy-preserving programs, and go home with the satisfaction of having tried our best. But just as compilation techniques have gotten more sophisticated in the past decades, cryptographers have simultaneously built multiparty cryptosystems with increasingly complex capabilities—capabilities that are incredibly difficult to directly reason about and take advantage of, as depicted along the vertical axis of Figure 1.1. To get around this, modern cryptosystems often provide simpler abstractions for programmers to use instead of directly interacting with the underlying protocols. A classic example is *programmable bootstrapping* (PBS): an operation that some MPC schemes support that allows directly evaluating a certain class of functions on encrypted values. Manually mapping computations down to these functions turns out to be quite challenging, so most implementations of PBS cryptosystems also provide a pre-built *library* of operations (usually a handful of Boolean gates) for programmers to use, thus trading off flexibility for programmability. However, as we discuss in more detail in Chapter 6, this greatly undersells the cryptosystem's capabilities. Indeed, by designing abstractions for the *compiler* instead of the *programmer*, we can avoid giving up this flexibility and still generate better code.

Compiler Cryptosystem Co-Design

Generalizing the insight from above, we arrive at the central premise of this dissertation: It is not enough to simply think of the cryptosystem as a fixed instruction set to target, nor is it enough to treat the compiler as a naïve translator from high-level programs to MPC operations. Instead, we need to build *MPC-aware optimizing compilers* while simultaneously *tweaking the protocol* to provide better abstractions for our MPC compilers, a philosophy we refer to as *compiler cryptosystem co-design*, and which fills in the upper-right corner of Figure 1.1. In this dissertation we explore what compiler cryptosystem co-design looks like for a particular class of MPC protocols known as *Fully Homomorphic Encryption*, or FHE.

We start by describing existing languages and compilers for MPC, and then present a number of novel works that use the philosophy of compiler cryptosystem co-design to improve on the current state-of-the-art.

1.1 Languages & Compilers for MPC

To understand and appreciate the novelty of our contributions in compiler cryptosystem co-design, it is essential to first survey the broader landscape of languages and compilers for MPC. Much of the early work in this space (e.g. Wysteria/Wys* [2, 3], HACCLE [14], Viaduct [15]) focuses not on performance, but on how to *express programs in MPC in the first place*. These works develop techniques to "extract" secure subcomputations based on information flow requirements, reason about complicated protocol-specific capabilities to select the correct MPC backend for each subcomputation, and verify both the correctness and security of the final generated code (e.g. by reasoning about the composability of the different protocols used). Such techniques serve to alleviate much of the burden of true "multiparty" programming, in which "mixed-mode computations" (ones that switched between secure and insecure phases, and between different MPC protocols) are not only possible, but unavoidable.

The early FHE space looks vastly different. Practically all FHE schemes consist of only two parties: A "client" encrypts their data with a private key and sends the ciphertexts to an "evaluator," who uses a public "evaluation" key to perfom homomorphic computations. Multiparty schemes do exist, but largely work via a "key exchange" protocol in which multiple clients collaborate to agree on a single evaluation key, before sending it to the evaluator along with their ciphertexts and proceeding as normal. Thus, reasoning about complicated information flow requirements is largely not required. Together with the fact that "scheme-switching" in the style of CHIMERA [16] and PEGASUS [17] was still several years away at this point, this means that the early developments in MPC languages didn't necessarily translate into better languages for FHE.

Instead, FHE compiler developers begin to focus on an orthogonal set of problems; ones that affect not correctness, but performance. A major challenge with optimizing FHE pro-

grams is how *opaque* the cost models are: Forgetting to relinearize a ciphertext may not affect the result of a computation, but it *can* cause the program to slow to even more of a crawl than before. As a result, we begin to see compilers that focus on *good automatic translation*, allowing programmers to write the logic of their application without worrying about such low-level details. Seminal works in this space include CHET [18], which automatically selects appropriate ciphertext layouts for tensor programs; EVA [10], which reasons about complicated cost models to insert important ciphertext maintenance operations like bootstrapping, rescaling, and relinearization; and Ramparts [1] and ALCHEMY [9], which supply programmers with efficient implementations of higher-level homomorphic kernels, and handle tedious tasks such as parameter selection. Crucially, although these techniques *do* significantly improve performance, they perform very few other optimizations, and essentially preserve the original structure of the program.

Recent years mark the emergence of compilers that finally cross this barrier to perform nontrivial optimizations on their source programs. These compilers can be classified into two camps: those that perform optimizations *automatically*, and those that *require some programmer effort*. In the first camp are works like the TFHE Transpiler [19] and its successor, the HEIR framework [20], which use hardware synthesis tools like XLS [21] and Yosys [22] to generate efficient FHE circuits and leverage the power of MLIR [23] to perform a number of standard optimizations; Fhelipe [24], which determines good data layouts for a program written in a Numpy-style DSL; and Concrete [25], which performs a number of rewrites on the computation graph extracted from a python program before generating code. In the other camp are works like Porcupine [26], which vectorizes FHE computations but requires the programmer to provide a *sketch* to constrain rotation patterns; and a number of DSLs like HECO [20] and Airduct [27] that require the progammer to express their computations in a restrictive array language to assist in vectorization.

1.2 Contributions

The ideas in this dissertation are organized into four chapters, each of which uses compiler cryptosystem co-design to tackle a different problem in the world of secure computation:

- We start by addressing the problem of FHE-aware vectorization in Chapter 3. Certain FHE schemes support a technique allows computations to be expressed over *packed ciphertext vectors* (Section 2.3.1) rather than individual ciphertexts. However, the unique semantics and cost models of these schemes mean can render traditional (SLP-based) approaches to vectorizing ineffective, often causing them to *degrade performance* rather than improving it. We present Coyote, the first compiler that FHE-specific cost models and heuristics to vectorize FHE programs. We show that Coyote consistently outperforms traditional SLP-style techniques, and even often discovers schedules that match the best expert-written implementations for some kernels.

- While Coyote considers arbitrary straight-line programs, in Chapter 4 we shift our attention to the special case of vectorizing *secure control flow* in the form of *decision forests* (Section 2.2). We present COPSE, an abstraction that sits on top of existing FHE cryptosystems and encodes the control flow of a decision forest as a series of highly vectorizable matrix operations. We demonstrate that using the COPSE abstraction to vectorize decision forests can accelerate inference by up to an order of magnitude.

- Chapters 3 and 4 consider vectorizing straight-line code and control flow *separately*, but real programs often contain *both*. In Chapter 5 we investigate how to combine these ideas in COIL, a language for expressing secure computations that contain both straight-line code *and* ciphertext-dependent branches. Along the way, we develop the idea of *path forests*, a novel intermediate representation for secure branching, and show how path forests unlock a number of optimizations that would not otherwise be possible with the usual *multiplexing* strategy.

- In Chapter 6 we finally depart from talking about vectorization and control flow to explore the depths of compiler cryptosystem co-design with COATL. Where COPSE built abstractions *on top of* existing cryptosystems, COATL *changes the programming abstraction the cryptosystems provide* and then *develops novel compilation techniques to target the new abstraction*. The particular cryptosystems we consider in this chapter are *Boolean FHE schemes*, in which ciphertexts are encryptions of bits, and computations

correspond to *homomorphically evaluating Boolean gates*. We present the *arithmetic lookup table*—a cryptographic abstraction that replaces Boolean gates—and develop compilation techniques to generate circuits built from these tables. We demonstrate that our techniques can take full advantage of existing machinery to synthesize and optimize Boolean circuits, but still generate circuits up to 1.5× smaller than their *already-optimized* Boolean counterparts.

The remainder of this dissertation is organized as follows: Chapter 2 develops necessary background and formalisms, Chapters 3-6 present our main contributions as described above, and Chapter 7 concludes by discussing some possible future directions to explore.

2. BACKGROUND

We provide here most of the background necessary to understand the chapters that follow, rather than dispersing it throughout.

2.1 Vectorization

Single instruction, multiple data, or SIMD, is a way of amortizing the run-time complexity of a program by *vectorizing* it, or lifting its scalar computation to one that operates over packed vectors. To vectorize, we need to first find sets of isomorphic scalar instructions and then decide how to pack the scalar operands of those instructions into vectors before replacing all of them with a single vector instruction. In traditional SIMD, this process relies heavily on the presence of data-parallel loops in the original program. Unrolling the loop by a few iterations (usually four or eight) produces a set of isomorphic instructions, one for each unrolled iteration. These are then packed into vectors, with one iteration per vector slot, and lifted into vector instructions. Thus, a loop that performs a scalar computation N times can be lifted into one that performs a semantically equivalent vector computation $N/4$ times.

Superword-Level Parallelism (SLP) is a more general technique that does not rely on the presence of loop-based control structures in the program to find vectorizable instructions. SLP analyzes a whole sequence of scalar instructions at once, looking for sets of *available instructions* (instructions whose operands have already been scheduled) that are all isomorphic to each other. At each step, it picks such a set and packs its instructions together into a vector, scheduling them together.

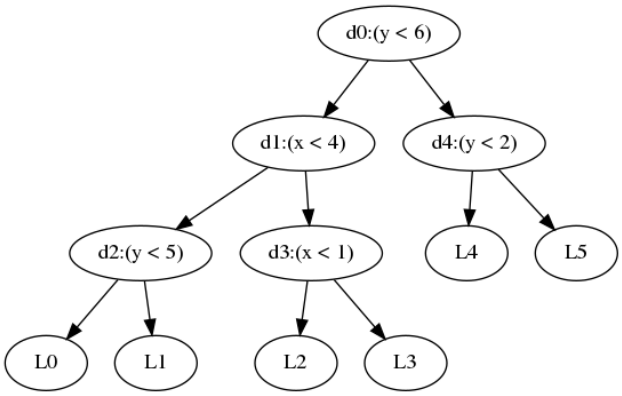


Figure 2.1. Example decision tree

2.2 Decision Forests

A *decision tree* is a classification model that assigns a class label to a vector of features by sequentially comparing the features against various thresholds. Figure 2.1 shows an example of a single decision tree. Inference over a decision tree is recursive. Leaf nodes of the tree correspond to class labels, whereas each interior "branch" node specifies a feature and a threshold. That feature from the vector is compared against the threshold, and depending on the result of the comparison either the left or right child of the tree is evaluated. For instance, the tree in Figure 2.1 uses x and y as its features and assigns class labels L_0 – L_5. Assuming the left branch is taken when the decision is false and the right branch is taken when true, the tree assigns the input feature $(x, y) = (0, 5)$ to the class label L_4. A *decision forest* model consists of several decision trees over the same feature set in parallel. Inference over a decision forest usually consists of obtaining a class label from each individual tree, and then combining the labels in some way (either by averaging or choosing the label selected by the plurality of trees, or some other domain-specific combining function).

2.3 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[1]. 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.

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.

Limitations

While FHE is a powerful technology that enables privacy-preserving computation, its lack of easy programmability prevents it from seeing widespread use. Because of the nature of secure computation, FHE does not support branching over ciphertexts—conditionals cannot depend on the values of encrypted data, otherwise the path taken through the computation leaks information about the data. In particular, this precludes FHE computations from having any kind of control flow structures, including conditionals and loops, that are control-dependent on ciphertexts. Applications must be expressed directly as *circuits* that transform ciphertexts using the homomorphic operations provided by the underlying FHE scheme: usually integer addition and multiplication for arithmetic schemes, or a fixed set of Boolean gates for Boolean schemes.

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

[1]↑Some authors have proposed multiparty multi-key extensions to FHE [28–30]. While we do not directly make use of these, the ideas in this dissertation are relatively straightforward to extend to a multikey setting.

expensive to evaluate on any nontrivial inputs; Writing efficient FHE programs requires a great deal of cryptographic expertise.

2.3.1 Ciphertext Batching

One way to mitigate the overhead of encrypted computation is via *ciphertext batching*, an optimization that certain (RLWE-based[2]) FHE schemes provide [31, 32]. 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).

This style of vectorization has a few peculiarities that distinguish it from normal vectorization:

1. The vectors are much larger than traditional hardware vector registers (e.g. several thousand slots wide, compared to the usual 4 or 8 slots). Utilizing this much space poses unique challenges.

2. Unlike with physical vector registers, there is no *indexing* primitive that can directly access a value in a particular slot of a ciphertext vector.

3. In general, the only way to move data between vector slots is by cyclically permuting its contents. This makes it much more important to assign vector lanes to packed instructions optimally, since realizing arbitrary permutations by composing several rotations quickly gets computationally expensive.

The challenges posed by points (2) and (3) in particular preclude directly using SLP-style vectorization, since its local reasoning means it does not sufficiently consider the high cost of data movement between lanes when deciding what instructions to pack together. Chapter 3 discusses this in more detail.

[2]↑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 chapter) is based on the hardness of RLWE.

2.3.2 Bootstrapping

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* [33, 34] 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.

2.3.3 CGGI

The CGGI scheme [35] 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: a 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.

A key observation we make in Chapter 6 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 6.2.1 provides a more formal treatment of this fact.) It is precisely this additional flexibility that we exploit in Chapter 6 to build more complex lookup tables and hence create circuits with fewer gates. Note that the number of gates in the circuit is the main metric that matters for performance: while programmable bootstrapping is a more lightweight technique than traditional (batched) bootstrapping, it is still far more expensive than any other CGGI operation. Thus, the latency of a CGGI circuit is almost entirely determined by the number of bootstraps—and hence, the number of gates—it contains.

2.4 MLIR/HEIR

MLIR[23] 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[36] 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 Chapter 6 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 [37]. 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 in Chapter 6.

2.5 Circuits & Multiplexers

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[3]. 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.

[3]↑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.

3. COYOTE

> The Coyote is limited, as Bugs is limited, by his anatomy.
>
> Chuck Jones

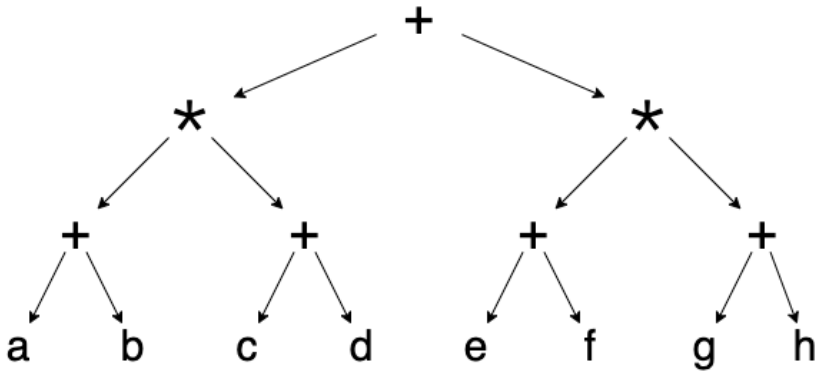


Figure 3.1. An example of an arithmetic circuit

In this chapter we develop a vectorization technique that is "FHE-aware"; That is, it understands the unique cost models and semantics of ciphertext packing. A natural first question to arise is: Why do we need an FHE-aware technique at all? Techniques like Superword-Level Parallelism (SLP) have existed for decades [13] and perform remarkably well in the classical vectorization space. Unfortunately, a naïve application of these techniques to FHE circuits fails. SLP aggressively packs isomorphic instructions into vectors, assuming that shuffling vector lanes around or indexing into a vector is relatively cheap. While this is a reasonable heuristic when targeting a physical vector architecture, FHE vectors are not hardware registers with slots for data; instead, the only way to move data between "vector lanes" is by performing a cyclic rotation of the entire vector. Realizing the shuffles incurred by SLP with a series of masks and rotations is expensive, and can quickly outweigh any benefits from vectorizing, as demonstrated below.

```
%0 = a + b
%1 = c + d
%2 = %0 * %1
%3 = e + f
%4 = g + h
%5 = %3 * %4
%6 = %2 + %5
```

(a) No vectorization

```
[%0, %1, %3, %4] = [a, c, e, g] +
                   [b, d, f, h]
[%2, _, %5, _] = [%0, _, %3, _] *
                 [%1, _, %4, _]
[%6, _, _, _] = [%2, _, _, _] +
                [%5, _, _, _]
```

(b) Aggressive vectorization, incurs two rotates

```
[%1, %3] = [c, e] + [d, f]
[%0, %4] = [a, g] + [b, h]
[%2, %5] = [%0, %3] * [%1, %4]
[%6, _] = [%2, _] + [%5, _]
```

(c) Optimal schedule, incurs one rotate

Figure 3.2. Possible schedules for Figure 3.1

The Vectorization/Rotation Tradeoff

Consider the arithmetic circuit[1] in Figure 3.1 implementing $((a + b) * (c + d)) + ((e + f) * (g + h))$. Applying SLP yields the schedule in Figure 3.2b, which packs together the four additions at the first level, and the two multiplies at the second level. The resulting schedule has a vector add, followed by a vector multiply, followed by an add. Rotations are needed between each operation to align the outputs of each operation with the next. Using an approximate model[2] of the relative latencies of each instruction in which multiplies and rotates have a latency of 1 and addition has a latency of 0.1, the total cost of this schedule is 3.2. However, by doing no vectorization and executing the circuit entirely with scalar operations (Figure 3.2a), we have five adds and two multiplies, with an overall cost of 2.5. In this case, vectorization actually makes the performance *worse*! The schedule in Figure 3.2c shows how we can do better: We pack the $a + b$ and the $e + f$ adds separately from the $c + d$

[1]↑We adopt the FHE-standard representation of arithmetic circuits as the intermediate representation for our programs [1, 26, 33, 38].

[2]↑Algorithms in HElib [39] assigns a "high latency" to both multiplies and rotates and a "low latency" to adds. For both simplicity and concreteness, we assume a 10:1 ratio between "high latency" and "low latency."

and $g + h$ adds, so that neither of them require a rotation to align with the multiply above them. By saving one rotation at the cost of an extra vector addition, we get a schedule with an overall cost of only 2.3. Clearly, we need a new vectorization strategy, and in particular, one that can properly account for the high cost of data movement throughout the program.

Co-optimization of Vector Packing and Data Layout

The main obstacle to optimality when using the classical SLP approach comes from the high cost of permuting vector elements: Aggressively packing instructions into vectors can require substantial and complex data movement to align operands for downstream vector instructions (e.g., SwizzleInventor [40] resorts to sketch-based synthesis to generate the appropriate permutations); Thus, such a schedule can incur so much overhead that no amount of vectorization makes it worth it.

More recent takes on SLP, such as VeGen [12] and goSLP [11], recognize the need to take the cost of data movement into account. VeGen, for instance, can decide to not pack certain instructions together because the data movement cost incurred is not worth it. However, VeGen does its reasoning *locally*; that is, it cannot reason about the effect packing instructions together may have on shuffling costs much later in the program. This tradeoff, fine in circumstances when shuffling is relatively cheap, is inappropriate for FHE, where shuffling is very expensive. While goSLP does reason globally, the cost model it uses to avoid over-packing is incompatible with the semantics of FHE vectorization. We discuss both of these techniques further in Section 3.5.

Our key insight is that because rotations are so expensive, data layout and vector packing are *fundamentally intertwined*; rather than treat these as separate problems, we must optimize them *together* when finding a schedule. In this chapter, we develop an approach that works at the level of *subcircuits*, splitting the program up into smaller pieces within which all the computations are locked into a single lane to avoid doing any rotations at all. While vectorizing across subcircuits gives up some packing potential (because operations within a subcircuit cannot be vectorized together), the savings on rotation costs can make up the difference: the subcircuits prevent over-vectorization that incurs too many rotations.

The optimal schedule of Figure 3.2c can be viewed as grouping $(a + b)$ with its downstream multiply in one subciruit, and $(g + h)$ with its downstream multiply in another subcircuit, and then vectorizing those two subcircuits together.

This approach yields a natural question: how do we decide which computations to merge into a subcircuit? This seems circular: subcircuit merging is intended to yield fewer rotations, which are determined by data layout, and data layout is driven by which operations are vectorized together, which in turn is constrained by subcircuit identification.

Contributions

We present Coyote, the first FHE-aware compiler that automatically vectorizes arbitrary arithmetic circuits. Coyote breaks the circular dependence between vector packing and data layout by using an iterative process that alternates between making packing decisions and determining data layout. Coyote uses simulated annealing to find optimal data layouts, and uses these to guide a best-first search towards optimal vector packs. Crucially, Coyote uses layouts from previous iterations of scheduling to identify subcircuits that would be profitable to merge, and then re-schedules based on the new subcircuits.

The specific contributions we make in this chapter are are:

1. An algorithm for simultaneously searching the space of data layouts and the space of vector packings to find an efficient combination.

2. A lightweight Python embedded DSL called Coyote, with a compiler that uses this algorithm to generate efficient FHE code for arbitrary programs

We test Coyote by using it to compile six computational kernels (matrix multiply, point cloud distances, 1D convolution, dot product, sorting a list, and finding the maximum element of a list), and compare the performance of the vectorized code to to the original unvectorized code. We also randomly generate several irregular polynomial-evaluation programs to measure the effect of things like operation density on Coyote's ability to vectorize. We find that Coyote yields efficient vector schedules with optimized rotations, and often recovers known-optimal expert implementations for certain kernels.

3.1 Coyote Overview

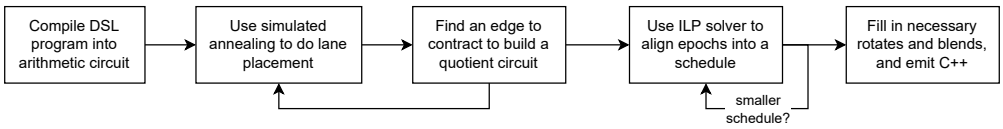
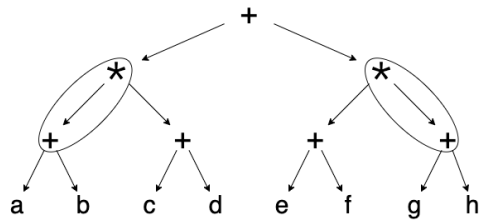
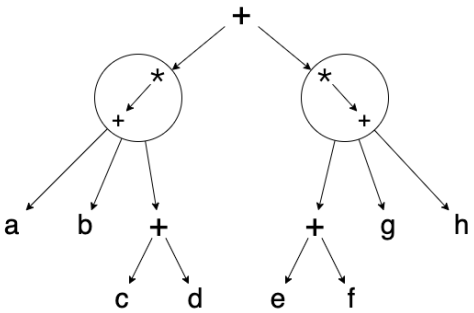


Figure 3.3. High-level compilation steps



(a) Circuit with subcircuits identified



(b) Quotiented circuit

```
[%1, %3] = [c, e] + [d, f]
[%0, %4] = [a, g] + [b, h]
[%2, %5] = [%0, %3] * [%1, %4]
[%6, _] = [%2, _] + [%5, _]
```

(c) Vector schedule after alignment

```
v0 = [c, e] + [d, f]
v1 = [a, g] + [b, h]
t0 = blend(v1@10, v0@01)
t1 = blend(v0@10, v1@01)
v2 = t0 * t1
s0 = v2 >> 1
v3 = v2 * s0
```

(d) Generated vector code

Figure 3.4. A running example of how Coyote vectorizes arbitrary arithmetic circuits

Coyote provides an embedded DSL (eDSL) that allows programmers to use a high level language to express computations in FHE. This computation is translated into an arithmetic circuit representing the computation, which is then compiled into vectorized FHE code. The process of compiling a circuit into vectorized code is shown in Figure 3.3, and described in more detail below, using the circuit in Figure 3.1 as a running example.

3.1.1 Compilation

1. Coyote *quotients* an input circuit (collapses subcircuits into single vertices) and assigns lanes to resulting vertices to produce a *protoschedule* that can be realized into a more efficient vector program. The result is a graph whose vertices correspond to connected subgraphs of the original circuit, such that no two vertices at the same height have the same lane (and hence are eligible to be vectorized together). Coyote collapses a subcircuit when it determines that the overhead of internally vectorizing it is not worth the gain from vectorization, so this step essentially forces certain operations to happen in scalar on a single lane. Section 3.2.2 describes how Coyote makes this decision.

   In the example in Figure 3.4a, the circled pairs of vertices are collapsed, yielding the quotient circuit in Figure 3.4b. The lane assignment for this protoschedule puts each un-quotiented addition on the same lane as its quotiented parent, and chooses one of these lanes on which to place the root of the tree.

2. The (collapsed) vertices at each height are aligned to pack together isomorphic nodes, producing a vector schedule from the protoschedule. In the example, the two adds at height 1 get trivially aligned, and the two "supernodes" at height 2 get aligned by packing together the two adds and the two multiplies. No alignment is needed for the single vertex at height 3. The details of the alignment procedure are given in Section 3.2.4. Figure 3.4c shows the result of this alignment.

3. Coyote compiles the schedule into a vector IR. The crux of this compilation step is figuring out when to *blend* and *rotate*. When a vector operand requires values from several different instructions, Coyote emits code to "blend" the results together into a single vector. When the lane an operand is used in is different from the lane it was produced in, Coyote emits a rotation instruction to move the operand into the correct lane. Notice that each arc in the protoschedule connecting vertices of different lanes corresponds to a rotation in the generated vector IR. Figure 3.4d shows the vector code Coyote generates for our running example. Notice that the generated code contains

two blends and one rotate. The blends are necessary[3] because on line 3 of the schedule, %0 and %3 are used in the same vector despite being produced in two separate vectors. Since none of the operands need to shift lanes, the vector instruction t0 = blend(v0@10, v1@01) takes [%0, %4] and [%1, %3] and blends them together to produce [%0, %3], which is exactly the operand used on line 3. Coyote emits a rotation because %5 gets used on a different lane than it is produced. The vector instruction s0 = v2 ≫ 1 takes [%2, %5] in v2 and produces [%5, %2] in s0. Section 3.3.1 describes the specifics of code generation.

```
def dot(v1, v2):
    return sum([a * b for a, b in zip(v1, v2)])

@coyote.define_circuit(A=matrix(3, 3), b=vector(3))
def matvec_multiply(A, b):
    result = []
    for i in range(len(A)):
        result.append(dot(A[i], b))
    return result
```

Figure 3.5. Coyote program for multiplying a vector by a matrix

3.1.2 DSL

A programmer can use Coyote's DSL (shown in Figure 3.5) to specify a computation and generate an arithmetic circuit. The DSL exposes a number of ways to annotate inputs to the computation: *replication*, *packing*, and fixing a *layout*. Annotating an input with "replicate" indicates that a copy of the input should be passed to the circuit for each place it is used (ensuring that each copy gets used exactly once). By default, inputs are unreplicated, meaning that an input that gets used in multiple places will have a fan-out corresponding to its usage frequency.

Specifying a "packing" constraint for a set of inputs requires that they be packed into a single input vector in the final circuit (note that inputs in the same vector are necessarily in

[3]↑We could, of course, simply exchange the positions of %3 and %4 and elide the blends. While Coyote *does* automatically perform this rewrite, we leave the blends in here for the sake of example.

different lanes). For example, a packing constraint might require that each entry of a matrix be placed in the same input vector, or that each row be placed in a unique vector.

After Coyote vectorizes the circuit as described above, it automatically packs the circuit inputs into vectors (while satisfying any provided packing constraints) and chooses the data layouts within these vectors. Alternatively, the programmer can choose to override this and manually provide an input layout. This is useful, for example, when composing multiple circuits, as the output layout of one determines the input layout of the next. The details of how these choices are made are discussed in Section 3.2.5, and the tradeoffs these annotations provide are discussed in Section 3.3, and evaluated in Section 3.4.7.

3.1.3 Backend

Coyote targets the BFV backend [4] for Microsoft SEAL [41]. The encryption parameters are hardcoded, and are chosen to allow for 8192 vector slots and a standard 128 bits of security.

3.2 Vectorization Procedure

When vectorizing arithmetic circuits with an SLP-style approach, at each step, we look at all available scalar instructions (whose source operands have all been scheduled), pick the largest set with the same operation, and schedule them together. This naïve strategy makes no guarantees about values being computed and used on the same lane; in other words, lining the computation up on incurs arbitrarily many shuffles. Unlike in normal vectorization, where applying arbitrary permutations to the lanes is relatively cheap, in FHE we are only allowed to rotate the entire vector by a fixed number of slots, and this rotation operation is expensive. Hence, the cost of bookkeeping quickly outweighs whatever benefit we might get from vectorization, making this approach not worth it.

When trying to vectorize an FHE program, we have two optimization problems to solve: instruction scheduling, and data layout. Optimizing only for instruction scheduling gives

[4]↑We could have instead chosen to use the CKKS backend, but BFV's cost model is more amenable to general vectorization. In particular, an operation we use often is "blending" slots from several vectors into one; while this is almost free in BFV, the cost of doing this in CKKS is nontrivial.

us the SLP approach: aggressively pack together isomorphic instructions without worrying about the incurred data movement overhead. Optimizing for data layout places us on the other end of the spectrum: to avoid having to do any rotates, we must place each connected component of the circuit on a single lane, precluding any vectorization and forcing us to execute everything as scalar operations.

One of our key insights is that these two problems are highly related, so we have to solve these *simultaneously* rather than independently, attempting to choose an optimal point in the tradeoff space between the two ends of the spectrum. In the following sections, we lay out the exact optimization problem as well as how we search for a solution.

3.2.1 Overview

The input to the compilation process is an arithmetic circuit, represented as a directed acyclic graph (DAG), where each vertex corresponds to an operation (gate) in the circuit and the leaves (vertices with no children) correspond to the inputs, and there is an arc $v_1 \to v_2$ if v_1 consumes v_2. When a particular input is used multiple times in the circuit, it can either be represented as a single vertex with an incoming arc from every gate that consumes it, or it can be *replicated* into multiple vertices which each get consumed once. This choice is expressed by the programmer in the surface language (Section 3.3).

The vector *protoschedule*[5] is a labeled quotient of the original circuit graph, where each vertex represents a connected subgraph, and is labeled with an integer representing the lane the subgraph gets placed on, such that no two vertices at the same height are labeled by the same lane. The protoschedule is naturally bi-graded into *epochs*, or groups of (independent) vertices at the same height which get packed together into a single sequence of vector instructions requiring no data movement, as well as *columns*, groups of vertices assigned to the same lane representing computation that happens in a single thread with no internal vectorization.

It turns out that both of our extremes from earlier can be realized in this model. Aggressively vectorizing SLP-style can be recovered by assigning a trivial subcircuit to each

[5]↑This structure is referred to as a *protoschedule* to distinguish it from the actual vector schedule, which explicitly computes an alignment for sequences of instructions at the same level.

vertex of the quotient, and simply enumerating lanes across epochs. On the other end of the spectrum, we could instead quotient the circuit into the discrete graph of its connected components and assign each vertex an arbitrary lane; this graph has no edges and requires no rotations, but also precludes any vectorization within connected components.

Finding a good protoschedule then requires us to first compute a "good" quotient that trades off between these extremes, together with a lane assignment that somehow maximizes our ability to vectorize without incurring too many rotations. This is expressed in the search procedure Coyote uses when finding a vector schedule: an outer loop performs a best-first search over possible quotient graphs, and an inner loop uses simulated annealing on each quotient to find a good lane placement. The result of the search procedure is a quotient of the circuit and a lane placement for the quotient, which together minimize[6] the cost of the resulting vector schedule. The next section discusses this search procedure in more detail.

3.2.2 Schedule Search

Given a cost model, we use a two-layer optimization strategy to produce a schedule that has good packing properties without incurring too much data movement overhead.

Determining lane placement (Algorithm 1)

The inner layer uses simulated annealing to find an optimal lane assignment for a given quotient graph. The initial assignment is the naive one given by simply enumerating the vertices at each epoch. At each step of the algorithm, we generate a candidate solution by randomly choosing two columns and a subset of the epochs in them to swap, maintaining the uniqueness condition of the schedule. If the overall cost (as described in Section 3.2.3) of the candidate solution is lower than the original cost, it is accepted, and used as the starting point for the next round. If the candidate solution cost is *higher* than the original cost, it is accepted with a probability that varies negatively with the difference in cost, and is generally

[6] ↑relative to the other quotients and lane placements visited in the search

smaller in later rounds than in earlier rounds[7]. After a fixed number of rounds have elapsed (see footnote), this algorithm returns the best solution found so far.

Computing optimal circuit quotient (Algorithm 2)

The outer layer searches the space of quotients for a graph that admits a good lane placement without giving up too much vectorizability. Here, we use a priority queue to implement a simple best-first search. Each graph in the queue is assumed to already be equipped with an optimal lane placement, via the algorithm described above. At each step, a graph is dequeued, and a new candidate solution is generated by looking at its set of cross-lane arcs and choosing one to contract (removing the edge and identifying its endpoints into a single vertex). The contracted graph may not be acyclic, so we continue contracting cycles until it is (in effect computing the condensation). The candidate solution is then enqueued with its cost from the annealed lane placement. If there are more available arcs to contract, the original graph is enqueued again.

After a fixed number of rounds have elapsed, or once the queue is empty, the algorithm terminates and returns the best graph. Since each step of this algorithm involves an expensive call to the lane placement procedure, this runs for a much smaller number of rounds, usually between 150 and 200. In practice, this is enough to find highly efficient schedules.

The next section discusses what makes one graph quotient or lane assigment "better" than another, and how these tradeoffs are quantified in Coyote's cost model.

[7]↑We use a slow cooling schedule with initial temperature $T_0 = 50$ and cooling parameter $\beta = 10^{-3}$. The probability of accepting a move that increases the cost by Δ_c is $e^{-\Delta_c/T}$. The annealing is run for 20k rounds.

Algorithm 1: Lane placement

Algorithm `PlaceLanes(`*graph*`)`

- *lanes* $\leftarrow$ `InitialPlacement(`*graph*`)`;
- $T \leftarrow T_0$;
- *cost* $\leftarrow$ `Cost(`*lanes, graph*`)`;
- **for** i $= 1 : N$ **do**
  - $T \leftarrow T/(1 + \beta T)$;
  - *candidate* $\leftarrow$ `GenerateCandidate(`*lanes, graph*`)`;
  - *cost′* $\leftarrow$ `Cost(`*candidate, graph*`)`;
  - **if** `Accept(`*cost, cost', T*`)` **then**
    - *lanes* $\leftarrow$ *candidate*;
    - *cost* $\leftarrow$ *cost′*;
- **return** *lanes, cost*

Procedure `Cost(`*lanes, graph*`)`

- $rotations[*] \leftarrow \emptyset$;
- **foreach** $(u \rightarrow v) \in graph$ **do**
  - **if** $lanes[u] \neq lanes[v]$ **then**
    - $rotations[u.epoch] \leftarrow rotations[u.epoch] \cup \{lanes[v] - lanes[u]\}$;
- $instrs[*] \leftarrow 0$;
- **foreach** $epoch \in graph$ **do**
  - **foreach** *opcode* **do**
    - $instr[opcode] \leftarrow instr[opcode] + \max_{col} count(epoch, col, opcode)$;
- **return** $w_R \times \sum_{ep} rotations[ep] + \sum_{op} w_{op} \times instrs[op]$

Algorithm 2: Computing a good circuit quotient

```
Algorithm ComputeQuotient(graph)
    lanes, cost ← PlaceLanes(graph);
    best ← lanes, graph;
    bestcost ← cost;
    pqueue ← [];
    Enqueue(pqueue, (graph, lanes), cost);
    for i = 1 : N do
        graph, lanes ← Dequeue(pqueue);
        if arc ⇜ CrossArcs(graph) then
            candidate ← Condensation(ContractEdge(graph, arc));
            lanes′, cost′ ← PlaceLanes(candidate);
            Enqueue(pqueue, (candidate, lanes'), cost');
            if cost′ < bestcost then
                best ← lanes′, candidate;
            Enqueue(pqueue, (graph, lanes), cost);
    return best
```

3.2.3 Cost Model

The cost of a particular protoschedule comes from two places: the number of rotations we have to perform, and the amount we have "given up" on vectorizing.

Rotations

Given a vector schedule, each *cross-lane arc* in the graph (an arc connecting vertices of different lanes) represents a rotation that must be performed to align an output from the tail of the arc to where it gets used at the head. However, determining the rotation overhead is not as simple as counting these arcs. Consider the case where instructions A and B are operands to instructions A' and B', respectively. If A and B are assigned lanes n and m, A'

and B' are assigned $n+k$ and $m+k$, and A and B end up packed together in the same vector instruction, the two separate data movement operations required for the $A \to A'$ arc and the $B \to B'$ can actually be performed by a single rotation by k (in fact, taking advantage of this fact is the main way Coyote optimizes data layout to require fewer total rotates). To compute the *actual* number of required rotations, we instead proceed epoch-by-epoch. For each epoch, we look at all cross-lane arcs with tails in that epoch, and compute the number of columns each spans (i.e. the required rotation amount) by subtracting the lane at the tail from the lane at the head. The rotation cost for that epoch is then just the number of distinct rotation amounts. For example, if a particular epoch has five cross-lane arcs, of which three represent a rotation of -1 and two represent a rotation of 6, its rotation cost is 2. It follows that the total rotation cost of a schedule is the sum of the rotation costs of each epoch.

Vectorizability

Taking successive quotients of the circuit reduces the total number of edges, and by extension, reduces the number of rotates that might be required; however, it also precludes any vectorization within the collapsed subcircuits. To account for this, we need a way of quantifying the amount of vectorization we are "giving up" with each quotient.

Unfortunately, directly computing the opportunity cost is very messy: the amount of vectorization we give up by identifying a set of vertices is not a property local to the vertices, but rather requires us to look globally at *all possible vertices* in those epochs, to see which vectorization opportunities are no longer available after the identification. Instead, we use an estimated *schedule height* as a proxy, with the justification being that giving up a lot of vectorization generally results in taller, less efficient final schedules.

The schedule height computation also proceeds epoch-by-epoch. For each epoch, we estimate the minimum number of vector instructions after packing by taking the maximum number of each type of operation across all the subcircuits associated to the vertices in that epoch. For example, the estimated height of an epoch containing one vertex with 3 adds

and 2 multiplies and another vertex with 2 adds and 4 multiplies would be 3 adds and 4 multiplies.

Overall Cost

The analysis presented above estimates the number of each type of instruction in the generated vector program. The final cost used a linear combination of all of these, with weights determined empirically by how expensive each instruction type is relative to the rest. In our implementation, we scale rotates and multiplies by 1, and addition and subtraction by 0.1.

3.2.4 Instruction Alignment

We align the instructions corresponding to the subcircuits in each epoch to produce a final vector schedule. It may seem like the solution to this is just sequence alignment, but aligning circuits is actually more complicated. At each step, the number of available children to align roughly doubles, meaning that the total number of subproblems to solve is exponential in the depth instead of linear. This causes the dynamic programming strategy of sequence alignment to quickly blow up.

Instead of wrangling so many subproblems, we can formulate this as an ILP. We create a variable for each scalar instruction representing its *schedule slot*, or the time at which it executes. We add constraints to require that each instruction be scheduled after all of its dependences, and also that two instructions with different operations never be scheduled at the same time. Finally, to speed up the search for a solution, we place a bound on the total length of the schedule which is iteratively tightened until the solver returns "unsatisfiable," meaning no shorter schedule could be found.

3.2.5 Data Layout

The circuit obtained after vectorization necessarily operates on inputs that have been "packed" into vectors. Choosing a good layout within these vectors is crucial, since a poor choice could incur many additional rotations to line operands up with where they are used.

Coyote can automatically select a good layout as part of the vectorization process. An input that is only used once is placed on the lane within its vector corresponding to the unique lane where it is used, and any two inputs that are placed on the same lane by this rule are packed into separate input vectors to avoid collisions.

For inputs that are used multiple times (or inputs that are required to be packed into the same vector, e.g. elements of the same matrix), Coyote places a no-op "load" gate in the scalar circuit (so that the input is only used once, by the load gate). Two load gates are placed in the same epoch in the circuit if and only if their corresponding inputs are required to be packed together (thus ensuring that they are given different lanes). The layout for these inputs is then determined by the lanes chosen for their corresponding load gates. This determines the data layout, as each input is placed on the same lane as its corresponding load gate (Section 3.1.2).

3.3 An eDSL and Compiler for FHE Programs

Coyote consists of an embedded DSL (eDSL) in Python that can be used to write FHE programs, shown in Figure 3.5. The DSL allows for arbitrary arithmetic computation over encrypted variables, and supports conditionals and loops over plaintext values. All conditionals and loops are fully evaluated and unrolled, and all function calls are fully inlined before generating the arithmetic circuit. The generated circuit is then passed to Coyote's back end, which vectorizes the computation as described in the previous sections, yielding a sequence of primitive vector operations that can be further lowered into C++ code targeting Microsoft SEAL's backend for BFV [41].

Coyote currently supports datatypes for encrypted inputs: scalar(), vector(size), and matrix(rows, cols). Inputs annotated with scalar() are free to be placed anywhere in the vector schedule; by contrast, matrix and vector inputs are always grouped together into vectors. The arithmetic circuits Coyote takes are directed acyclic graphs (DAGs) that fail to be trees exactly when values are used as inputs to multiple computations (e.g. the value a in $ab+ac$). Any such DAG can be turned into a tree by *replicating* inputs (Section 3.1.2) in a "reverse-CSE" process (for example, $ab + ac \to a_1b + a_2c$, where the value of a is supplied to both

a_1 and a_2). This results in circuits with better rotation characteristics at the cost of extra computation. By default, Coyote automatically replicates all scalar inputs, and leaves all vector and matrix inputs unreplicated, but this behavior can be overridden by the programmer (note that replicating a vector or matrix input results in multiple copies of each variable all being grouped into the same vector).

Automatically choosing a data layout

While specifying a set of inputs as a vector(n) or matrix(m, n), Coyote restricts the space of available schedules to the ones that group these inputs together. However, it is still free to choose a particular layout within the vector (i.e., $a[0]$ and $a[1]$ need not be placed in adjacent lanes[8]). In practice, this allows Coyote to choose a layout that minimizes the rotations required to align inputs with where they are used. We evaluate the effectiveness of this choice in Section 3.4.7.

3.3.1 Code Generation

The algorithm in Section 3.2 produces a vector schedule (i.e. a lane and schedule slot for each scalar, where the schedule slot determines the order in which instructions get executed). Coyote compiles this schedule to a simple vector IR by scheduling vector instructions according to a topological sort, inserting rotations as needed. The vector ISA supports vector addition, subtraction, multiplication, and rotation, as well as a constant load instruction and a *blend* instruction. The semantics of the blend instruction are a bit subtle: it mixes lanes from multiple vector registers into the same register *while keeping all data on its original lane*. For example:

$$blend(x_1x_2x_3x_4@1010, y_1y_2y_3y_4@0101) \rightarrow [x_1y_2x_3y_4]$$

[8]↑Note that a noncanonical input layout potentially means that two kernels compiled by Coyote may not be composable. We overcome this by allowing the programmer to manually provide their own layout (Section 3.1.2).

In the backend, the blend is implemented as a series of plaintext bitmasks followed by ciphertext adds.

3.4 Evaluating Coyote

In this evaluation, we aim to answer the following questions:

1. **How effective is Coyote's vectorization?** To answer this, we count the number of instructions generated in the vector code compared to the scalar code. (Section 3.4.2)

2. **How much speedup does compiling with Coyote get us?** To address this, we compile several realistic benchmarks and measure the speedup of the vectorized code over scalar execution. (Section 3.4.3)

3. **How well does Coyote scale up to larger kernel sizes?** To measure this, we compile a much larger circuit by vectorizing and composing its components. (Section 3.4.4)

4. **How well do Coyote's schedules compare to hand optimized code?** We compare the run times of various hand-optimized benchmarks to the run time of the vector code that Coyote generates for them. (Section 3.4.6)

5. **To what extent does the data layout chosen by the programmer affect Coyote's ability to vectorize?** We conduct a case study exploring various data layout schemes for a 3×3 matrix multiply, and measure the vectorization speedup for each layout. (Section 3.4.7)

6. **How effective is the layout/schedule co-optimization strategy?** We track the progress of the schedule-search procedure over time for various levels of data layout optimization. (Section 3.4.8)

7. **How much optimality do we sacrifice by setting synthesis timeouts?** We turn off the synthesis timeouts to guarantee optimal schedule alignments and compare the results. (Section 3.4.9)

3.4.1 Computational Kernels

To assess Coyote's ability to vectorize general applications, we use it to compile a suite of benchmarks and measure the speedup from vectorization. While there is not currently a standard benchmark suite on which to evaluate FHE-based compilers, we choose a set of benchmarks similar[9] to those used by Porcupine [26], representing a spectrum of both regular and irregular computations, as well as ones that are both sparse and dense in terms of data reuse.

The benchmarks are as follows:

1. Multiplying two matrices. We do this with 2×2 matrices (regular, little data reuse) and 3×3 matrices (regular, some substantial data reuse).

2. Vector dot product, with vector sizes of 3, 6, and 10 (all of these are regular with no data reuse).

3. 1D convolution. We do this with a vector of size 4 and a kernel of size 2, and with a vector of size 5 and a kernel of size 3. Both of these are regular and have little data reuse.

4. Point cloud distances (Given a set of points, compute the square of every pairwise Euclidean distance). We do this for 3, 4, and 5 points. These are all regular but have some substantial data reuse, especially in the 5-point case.

5. Sorting a list of size 3. This benchmark implements the sort as a "decision tree," taking as input three ciphertexts representing pairwise comparison results and six ciphertext "labels" representing possible arrangements of the sorted list. In particular, each of the three comparison results gets used in multiple branches of the tree. "grouped" here means the data layout groups the three comparisons into one vector and the six labels into another. This is irregular, and the grouped versions have data reuse.

[9]↑While we do not have access to Porcupine's actual benchmarks for a direct comparison, their polynomial regression corresponds to our matrix multiply, their L2 distance correpsonds to our point cloud distance, and most of their image processing kernels are specific convolutions (i.e. plaintext kernels), while we evaluate on generic convolutions. The input sizes of our benchmarks are comparable to those of Porcupine's.

6. Finding the maximum element in a list of size 5. This benchmark takes as input five ciphertexts representing the elements of the list, and ten ciphertexts representing pairwise comparison results. Similar to the sorting benchmark, this is irregular, and the grouped versions have data reuse.

To investigate the effect of data replication we use three different replication strategies for each benchmark:

1. *Unreplicated*, where each input appears in only one input vector, and hence must be used by multiple operations.

2. *Partially replicated* in which one of the two inputs is fully replicated—and hence each operation that requires that input gets its own copy, obviating the need for data movement—while the other is not.

3. *Fully replicated* in which both inputs are fully replicated.

Note that for dot product, replication makes no difference as each input is used exactly once.

3.4.2 Costs and Effects of Vector Compilation

Table 3.1 shows benchmark properties, including how long each benchmark takes to compile and how many operations each program has before and after vectorization. Our benchmarks range in size from 5 scalar instructions (size 3 dot product) to 75 instructions (5 point distances)

While the number of scalar multiplies go as high as 27 (for the 3×3 matrix multiply), Coyote is almost always able to pack these into at most one or two vector instructions. The main exceptions to this rule are the highly irregular tree benchmarks, which still go from 10 scalar multiplies to between 4 and 5 vector multiplies. Another point to notice is the number of rotates. Most benchmarks require fewer than 10 rotates. However, the 5 point distance and 3×3 matrix multiply benchmarks have a very high number of rotations after vectorization, as these are the most data-dense ones—intermediate results are required by many downstream computations, incurring considerable overhead.

Table 3.1. Compilation time in seconds, as well as instruction counts in the scalar and vector code, and the ideal speedup (work/span).

Benchmark	Time	Scalar		Vector				Ideal
		Add+Sub	Mul	Add+Sub	Mul	Rot	Blend	
conv.4.2.un	97	3	6	2	1	4	4	5.73
conv.4.2.partially	80	3	6	2	1	2	1	5.73
conv.4.2.fully	71	3	6	2	1	1	2	5.73
conv.5.3.un	206	6	9	3	1	7	5	8.0
conv.5.3.partially	211	6	9	4	1	4	3	8.0
conv.5.3.fully	208	6	9	4	1	2	3	8.0
dist.3.un	228	18	9	1	1	4	6	9.82
dist.3.partially	233	18	9	1	1	4	6	9.82
dist.3.fully	226	18	9	2	2	2	7	9.82
dist.4.un	425	32	16	2	2	6	8	17.45
dist.4.partially	432	32	16	2	2	6	8	17.45
dist.4.fully	463	32	16	2	2	4	8	17.45
dist.5.un	619	50	25	2	2	13	10	27.27
dist.5.partially	629	50	25	2	2	13	10	27.27
dist.5.fully	609	50	25	2	2	9	10	27.27
dot.3.un	10	2	3	3	1	2	0	2.67
dot.3.partially	10	2	3	3	1	2	0	2.67
dot.3.fully	10	2	3	3	1	2	0	2.67
dot.6.un	159	5	6	4	1	3	2	5.0
dot.6.partially	154	5	6	4	1	3	2	5.0
dot.6.fully	156	5	6	4	1	3	2	5.0
dot.10.un	254	9	10	6	1	5	4	7.79
dot.10.partially	257	9	10	6	1	5	4	7.79
dot.10.fully	251	9	10	6	1	5	4	7.79
mm.2.un	176	4	8	2	1	3	2	7.64
mm.2.partially	171	4	8	2	1	2	1	7.64
mm.2.fully	170	4	8	2	1	1	2	7.64
mm.3.un	573	18	27	5	2	20	10	24.0
mm.3.partially	610	18	27	4	1	18	6	24.0
mm.3.fully	607	18	27	4	2	9	3	24.0
sort[3].grouped.un	238	10	10	8	4	8	6	3.24
sort[3].grouped.partially	233	10	10	8	4	4	3	3.24
sort[3].grouped.fully	231	10	10	8	4	4	3	3.24
sort[3]	139	10	10	11	5	1	1	3.24
max[5].grouped.un	880	30	30	15	11	30	18	7.33
max[5].grouped.partially	905	30	30	12	9	27	15	7.33
max[5].grouped.fully	902	30	30	19	9	24	23	7.33
max[5]	537	30	30	17	6	15	13	7.33
mm.16.blocked	7139	3840	4096	414	128	4446	872	3200

Furthermore, the unreplicated versions of each benchmark almost always incurs more rotates than the partially or fully replicated versions, validating our hypothesis that input replication helps alleviate the rotation burden.

Table 3.1 also shows the ideal speedup for each benchmark, using our cost model of multiplies being 10× as expensive as adds and subtracts. Note that this modeled speedup is based on a classic work/span analysis, and hence assumes that permuting and shuffling data between vectors is free. This modeled speedup thus represents a substantial overestimate of the actual speedup that could be achieved in the program. For example, a 3x3 matrix multiply has nine 3-element dot products. The necessary 27 multiplies can all be performed in one vector operation, but the results of three multiplies need to be added to perform each dot product. Thus in reality two of the multiplies are performed in the "wrong" lane and need to be shuffled to the correct lane to complete the computation.

3.4.3 Speedups

While instruction counts indicate that Coyote is able to effectively find vector operations for each benchmark, we must also determine whether the actual costs of rotations and blends outweigh the vectorization benefits. Hence, we run each benchmark 50 times in scalar, and 50 times after vectorization to compute the speedup from vectorization, shown in Figure 3.6. We find very little variance in execution time across individual runs for any benchmark. Each benchmark has three bars representing, in order from left to right, the unreplicated, partially replicated, and fully replicated runs. The sort[3] and max[5] benchmarks have an extra green bar representing the ungrouped run (without grouping, all inputs are fully replicated no matter what). We see speedups ranging from 1.5× on the data-dense point cloud distances benchmarks to over 3.5× on the highly vectorizable matrix multiply. We also generally notice more speedup as the replication level increases, suggesting that Coyote is able to take advantage of replicated inputs to eliminate rotations from the schedule.

While it may appear that Coyote's actual speedups are sometimes well off from the idealized speedups, this is due to the rotations and blends required outside an idealized world where vector permutation is free. For example, in the 3x3, fully-replicated matrix

multiply case, Coyote generates 9 rotations to move results into place. We see, though, that in benchmarks where Coyote can generate schedules with few rotates, it does well despite the data movement costs. For example, in conv.4.2, Coyote achieves a 2.5× speedup versus an ideal speedup of 5×; and in dot.3, Coyote achieves a 1.6× speedup versus an ideal speedup of 2.7×.

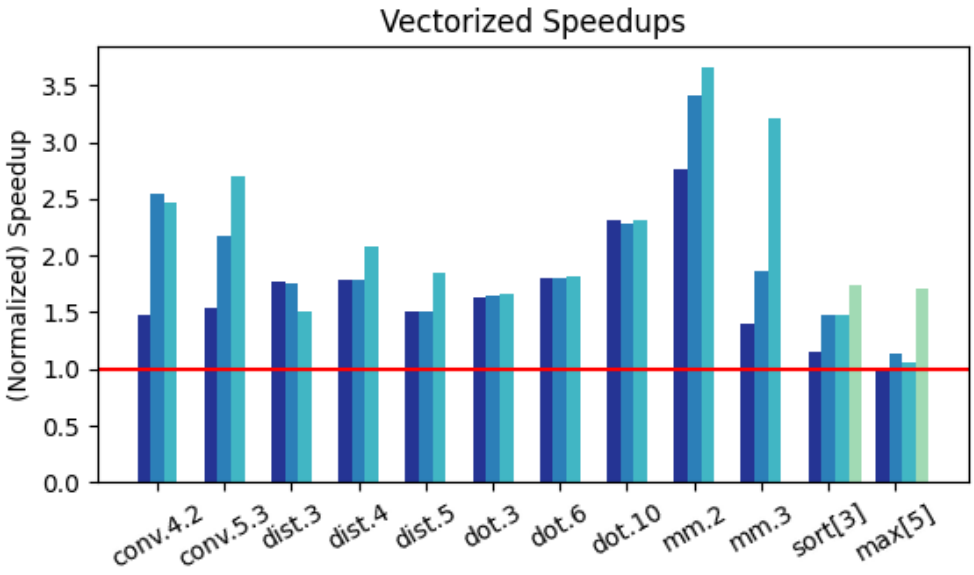


Figure 3.6. Speedup of vectorized code over scalar (higher is better). Left-to-right, the first three bars for each benchmark represent unreplicated, partially replicated, and fully replicated inputs, respectively. The fourth bar for the `sort[3]` and `max[5]` benchmarks represent ungrouped inputs.

3.4.4 Scalability

Many of the benchmarks we evaluate on have relatively small input sizes, since it is often intractable to directly apply the lane placement search procedure. However, it is possible to scale Coyote up to larger input sizes by "blocking," or vectorizing smaller kernels separately and then composing the vector programs. To investigate how well this works, we use Coyote

to compile a 16×16 matrix multiply as follows. We vectorize the multiplication of a single 4×4 "block," and record the input/output layouts Coyote chooses.

The output layout of each 4×4 block is used to fix the input layout to another kernel (see Section 3.1.2), which takes 64 of these blocks and performs the necessary reductions to arrange them into the final 16×16 matrix multiplication. The metadata for this benchmark is shown under `mm.16` in Table 3.1 (the compilation time includes the time to compile the 4×4 block as well as the reduction circuit). After vectorizing, the blocked 16×16 matrix multiply takes 3433 seconds, compared to 4541 seconds before vectorizing, for a total 32% speedup. This shows that by composing smaller kernels, Coyote is able to scale up to vectorizing much larger circuits and still see relatively significant speedups over unvectorized code. Note that this understates Coyote's potential speedup as it does not currently attempt to vectorize separate, identical kernels together (which is a regular process so could use standard vectorization techniques).

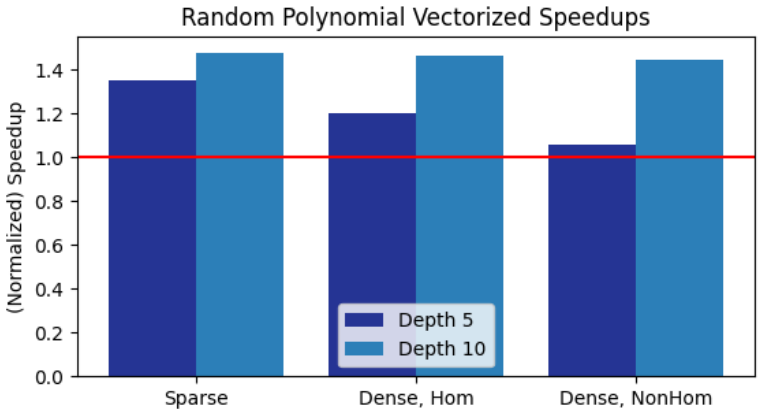


Figure 3.7. Speedups for random polynomials (higher is better).

3.4.5 Randomly Generated Irregular Kernels

To further investigate Coyote's ability to vectorize, even in the absence of a regular structure on the computation, we randomly generated several polynomials to evaluate as

arbitrary arithmetic expression trees. The trees are generated according to three different regimes to cover different kinds of programs:

1. *Dense, homogeneous*: The expression tree is both full and complete, and all the operations are isomorphic. In principle, this represents a best case for vectorization.

2. *Dense, nonhomogeneous*: The expression tree is both full and complete, each operation has a 50/50 chance of being an add or a multiply. Hence, while the trees are structurally similar, the heterogeneity of operations means that vectorization opportunities are restricted.

3. *Sparse*: Many operations have one leaf node input, the tree is not very balanced. In principle, this represents a worst case for vectorization, where Coyote must work hard to find vectorizable computation.

For each regime, we generate ten total polynomials, five with a circuit depth of 5 and five with a circuit depth of 10. Each polynomial is run 20 times in scalar and 20 times after vectorization, and we average speedups across the five polynomials in each regime before reporting. These speedups are shown in Figure 3.7. We see that Coyote is able to achieve speedups of up to 1.4× by vectorizing. Looking at the depth 5 dense nonhomogeneous polynomials, we find that many of them are too small and irregular to admit any profitable vectorization; in these cases, Coyote is correctly able to deduce that the scalar execution strategy is optimal rather than attempting to vectorize and incur spurious rotations. Since the generated vector code is identical to the scalar code for several of these, the average speedup is very close to 1.0.

We find that both sparse and dense homogeneous polynomials see substantial benefits from vectorization, with sparse polynomials having more speedup. This may seem surprising: dense homogeneous trees appear to be a best case scenario for vectorization, as all of the operations can be perfectly packed together. However, the key to this result is that rotations are expensive. The sparse trees have many vertices of arity 1—these operations do not require any rotations to align their input operands. In contrast, the dense trees require more rotations, canceling out the benefits from greater vectorization. This is further justifi-

cation for Coyote's design decision to focus on minimizing rotation in its schedule search. In the light of this discussion, it is perhaps *un*surprising that the dense trees (requiring more rotation) with non-homogeneous operations (limiting vector packing) ultimately have the lowest speedup.

3.4.6 Comparison to Hand-Optimized Schedules

To compare Coyote's vectorized schedules to hand-optimized baselines, we compile three kernels with Coyote: matrix/vector multiply, dot product, and point-cloud distance. Each of these kernels has a well-known expert-optimized baseline implementation, which we also implement in Coyote's vector IR, before compiling both to C++ and measuring the time it took to run each one 50 times. The results are shown in Table 3.2. For smaller sizes, we see Coyote's vectorization is capable of matching or even outperforming the expert-written baselines, although on larger sizes the search space is often too big to automatically find the expert schedules. Manually inspecting the generated code shows that this is usually because the schedule Coyote generates used one or two more rotates than the baseline. In the case of the dot product, the schedules Coyote finds all use the same number of rotates as the expert schedule, but occasionally incur more blends.

Table 3.2. Coyote vectorization vs. expert-written code

Benchmark	Coyote time (s)	Expert time (s)
mv.2	2.37	2.51
mv.3	3.3	3.9
mv.4	7.7	5.3
dist.3	3.5	3.2
dist.4	8.4	4.0
dist.5	15.3	5.5
dot.3	1.6	1.6
dot.6	2.9	2.2
dot.10	3.8	2.6

3.4.7 Effects of Data Layout

To study the effects of different data layout choices, we vary the data layout in 3×3 matrix multiply:

Together: The matrices A and B are grouped into a single vector of 18 elements

Separate: A and B are grouped into individual vectors (this is the normal layout used in benchmarking in Figure 3.6)

Rows/Cols: The rows of A are grouped into three separate vectors, as are the columns of B.

Cols/Rows: The columns of A are grouped into three separate vectors, as are the rows of B.

Individual: Each of the 18 entries are grouped into their own vector (note that this is different from simply leaving them as free scalars, because this precludes Coyote from choosing to put some of them on the same vector anyway).

In each layout, all the inputs are unreplicated. Figure 3.8 shows the results of this case study. Interestingly, we find that grouping the matrices together yields greater speedups than keeping them separate. When multiple entries are on one vector, Coyote can arrange the elements such that rotating one automatically gives useful rotations of the others. By contrast, when each entry is on a separate vector, every rotation must necessarily be done separately, so that schedule ends up with much more overhead. In particular, we find that `indiv` requires more than twice as many rotations as `together`.

3.4.8 Effects of Search and Co-Optimization

We would like to know how effective Coyote's schedule search is, and in particular, how good of a job it does at optimizing the data layout and schedule *together*. To test this, we compile the 5-point distances benchmark with three different levels of data layout by varying the number of iterations of simulated annealing, and record the cost of the generated schedule during each round of the search.

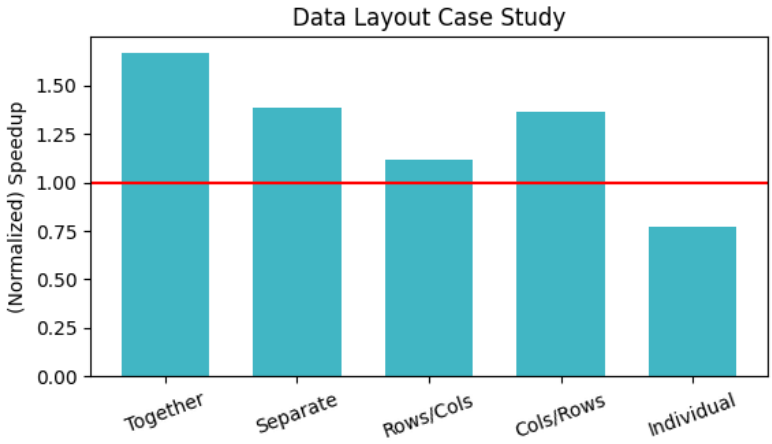


Figure 3.8. Speedups for the five data layout case studies (higher is better). Note that the second bar ("Separate") corresponds to the leftmost bar of mm.3 in Figure 3.6.

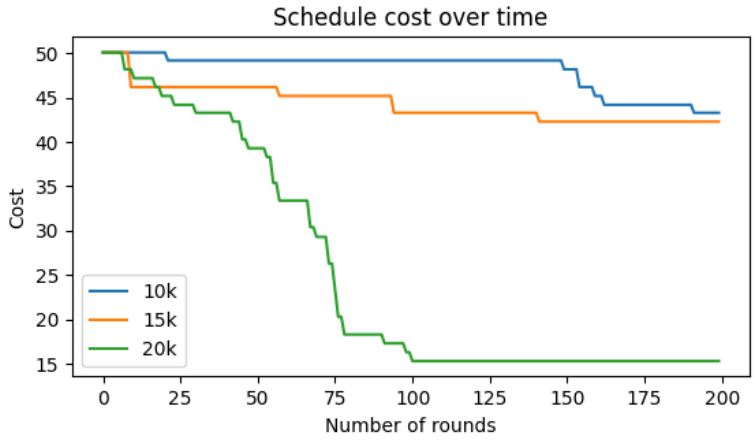


Figure 3.9. Schedule cost over time (lower is better) for different numbers of simulated annealing iterations for data layout per step of scheduling.

Table 3.3. Comparison of compilation time (seconds) and vectorization speedup with vs. without synthesis timeouts.

Benchmark	Timeout		No Timeout	
	Time	Speedup	Time	Speedup
dist.5.un	619	1.70	752	1.75
dist.5.partially	629	1.73	751	1.76
dist.5.fully	609	2.08	752	2.16
conv.4.2.un	97	1.75	122	1.78
conv.4.2.partially	80	3.12	102	3.19
conv.4.2.fully	71	3.21	87	3.15
conv.5.3.un	206	1.79	260	1.80
conv.5.3.partially	211	2.48	261	2.57
conv.5.3.fully	208	3.32	254	3.30
dot.3.un	10	2.11	13	2.15
dot.3.partially	10	2.11	13	2.13
dot.3.fully	10	2.12	13	2.14
dot.6.un	159	2.51	202	2.20
dot.6.partially	154	2.51	193	2.21
dot.6.fully	156	2.51	198	2.20
dot.10.un	254	2.21	320	2.75
dot.10.partially	257	2.15	323	2.77
dot.10.fully	251	2.27	311	2.74

Figure 3.9 shows the cost of the vector schedule over time for various levels of data layout. The blue line depicts the schedule cost when we use 10k iterations of simulated annealing to find an optimal data layout at each step; the orange line is 15k iterations and the green line is 20k iterations. As expected, doing more iterations of simulated annealing has a large effect on the efficiency of the final schedule, since we rely on the fact that the data layout being used to guide each round of the search is close to optimal. In compiling all our benchmarks, we use 20k iterations of annealing.

3.4.9 Optimality Tradeoffs from Timeouts

The synthesis procedure we use for generating the final (aligned) schedule from a proto-schedule uses iterative calls to an ILP solver that reduce the schedule height constraint until hitting a timeout. This is to prevent the synthesis time from blowing up, but it does come at the cost of sacrificing some optimality, since the solver might time out before finding the smallest possible schedule. In this section, we investigate the extent to which this choice matters by instead using a version of the scheduler that directly finds the minimal-height schedule with no timeout. This approach guarantees that the synthesized schedules have minimal height.[10] Table 3.3 summarizes the results. We see that compilation time increases with the optimal ILP, but speedups are comparable (sometimes faster, and even sometimes slower, due to different blends).

3.5 Other SLP Vectorizers

VeGen [12] is a recent variant of SLP, introducing a notion of *lane level parallelism* that encodes which lanes are performing which computations, allowing it to reason about rotation costs when building vector packs. For example, VeGen can reason about the rotation costs to pack together operands for an instruction into a temporary vector, and can use this to decide whether or not packing those instructions is worth it. However, this reasoning only happens locally, and VeGen does not incorporate information about how instruction packing might affect later rotations.

goSLP [11] reasons about globally optimal packing, and finds lane placements that minimize data shuffling costs. However, there are assumptions baked into its cost model that make it fundamentally unsuitable for the FHE setting. goSLP frames vectorization overhead in terms of the number of *pack* and *unpack* operations incurred. For example, permuting the slots of a single vector incurs one unpack, whereas blending the contents of N vectors (without any permutation) incurs N unpacks. This cost model implicitly requires wide blends to be more expensive than arbitrary permutations, almost the opposite of FHE's cost model.

[10] ↑Note that because the alignment algorithm cannot account for blend costs—they arise after alignment—the minimal *height* schedule may not be the minimal *cost* schedule.

In FHE, blends are almost free (instantiated as cheap plaintext multiplies and ciphertext adds) whereas a "bad" permutation can require O(n) rotates to realize. In other words, goSLP will often forego a highly profitable schedule with many blends and few rotates, and instead opt for a more conservative one. Additionally, goSLP does lane placement (permutation selection) *after* finding vector packs, creating situations like the one described at the beginning of this chapter, in which the ostensibly profitable packing does not admit a good data layout. By contrast, Coyote's cooperative scheduling strategy precludes this.

4. COPSE

> Not far from our young hero's peaceful home
> Lies the fair grove wherein he loves to roam.
> Tho' but a stunted copse in vacant lot,
> He dubs it Tempe, and adores the spot
>
> H.P. Lovecraft

In the previous chapter, we described a technique for adapting SLP-style vectorization to the unique semantics and cost models of FHE computation. While SLP-style approaches like Coyote excel at vectorizing straight-line code, in this chapter we turn our attention to a class of problems they struggle with: decision tree inference. Traditional algorithms for decision tree inference are inherently sequential, because the decision result at one branch influences the control flow path that inference takes, and thus which other branches to try (most modern SLP vectorizers choose to entirely avoid the problem of dealing with control flow, and operate at the basic block level instead).

However, decision trees in FHE, and in particular ones in which the *branching conditions* are private, look different: Because the party performing inference cannot access any private branching conditions, all control flow in the tree must be *linearized*. A common approach to control flow linearization is via the use of *multiplexers* (muxes), or special operations which first fully evaluate the condition and all possible results of a conditional before obliviously selecting the correct one.[1] This is equivalent to the approach taken by existing secure decision tree inference systems like Aloufi, et. al [42]: Replacing each secure branch with a mux transforms the decision tree into a large polynomial, which can then be evaluated using FHE primitives. Unfortunately, the polynomials produced by this naïve linearization are not easily vectorizable (e.g., on the sort benchmark from the previous chapter, which is implemented as a decision tree polynomial, Coyote achieves speedups of a mere 1.1×). Clearly, a linearization

[1]↑An example implementation of a mux encoding "if b then X else Y" is $bX + (1-b)Y$. Notice the mux evaluates b, X, and Y no matter what, whereas the conditional first evaluates b and then decides which branch to take.

strategy as unstructured as multiplexing is not well-suited to programs like decision trees; can we do better?

COPSE

In this chapter, we present COPSE, a cryptographic abstraction specialized to the problem of vectorizing decision tree inference. COPSE leverages FHE's "evaluate all paths" semantics to relax the control flow dependences in traditional decision tree inference, and restructures the sequential algorithm into one that more readily maps onto existing vectorized FHE primitives. The restructured computation occurs in four steps: a *comparison* step in which all the branches are evaluated (in parallel), a *reshaping* step in which branch decisions are shuffled into a canonical order, a *level processing* step where all decisions at a particular depth of the program are evaluated, and an *aggregation* step in which the results from each depth are combined into a final result.

In particular, this chapter makes the following contributions:

- A cryptographic abstraction that enables encoding decision forest inference in terms of easily vectorizable FHE operations
- A compiler that lowers decision forest models down to this abstraction, as well as a runtime that encrypts compiled models and executes secure inference queries
- An analysis demonstrating that our approach produces low-depth FHE circuits that make effective use of ciphertext packing

Additionally, conduct a sensitivity analysis and find that while there is a linear relationship between level processing time and the number of branches in the program, some of the work (such as the comparison step) is done only once and takes a constant amount of time regardless of the model size. We also train our own models on several open source ML datasets and show that our compiler not only generates FHE circuits that perform classification several times faster than previous work, it can also easily scale up to much larger models by exploiting the parallelism we get "for free" from this restructuring.

While there are FHE schemes that allow for three distinct parties (the server, the model owner, and the data owner), either by using multi-key constructions or by using some protocol to compute a secret key shared between the data and model owners, neither option is currently implemented in HElib (as of version 1.1.0). Therefore, we test our benchmarks on scenarios in which there are only two real parties: either the model and data owners are the same party offloading computation, or the server also owns the model and allows the data owner to use it for inference.

4.1 COPSE Overview

This section gives a high level overview of the vectorizable decision forest evaluation algorithm before diving into the details. We will use the decision tree in Figure 2.1 as a running example to illustrate these steps.

4.1.1 The Players

The algorithm deals with three abstract entities: the model owner (Maurice), the data owner (Diane), and the server that performs the computation (Sally). These could correspond to different physical parties who want to conceal information from each other, or multiple entities could map to the same physical party (for example, the same party could own the model and the server). Section 4.6 discusses the security implications of different configurations. The three entities each own different components of the system, and work together to evaluate a decision forest for a set of features.

1. Maurice owns a decision forest model for which the features and labels are public, but he wants to keep the shapes of the trees and their threshold values secret

2. Diane owns several feature vectors that she wants to classify using the model owned by Maurice

3. Sally owns no data but possesses computational power and allows Maurice and Diane to offload their computations to her.

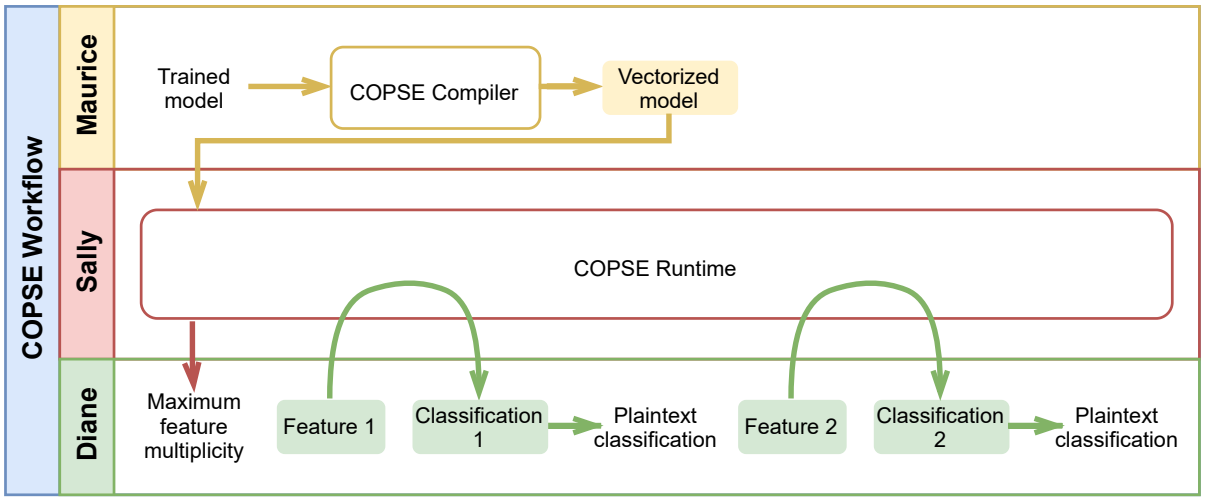


Figure 4.1. High-level COPSE system workflow. Yellow components and data are Maurice's responsibility. Red components are Sally's. Green components are Diane's. Shaded boxes represent encrypted data.

4.1.2 The Workflow

Figure 4.1 shows a high level overview of the COPSE workflow. The COPSE system consists of two main components: a compiler used by Maurice, and a runtime used by Sally.

Once Maurice has trained a decision forest model, he uses the COPSE compiler to generate an encrypted and vectorized representation that he can send to Sally, who can then accept inference queries, and use the COPSE evaluation algorithm (summarized next) to make classifications. When Diane wants to make a query, she first encrypts her features and then sends them to Sally, who uses the COPSE runtime to classify them against Maurice's encrypted model. Sally then sends the encrypted classification result back to Diane, who can then decrypt it and send additional inference queries to the model if she wishes.

4.1.3 The Evaluation Algorithm

The multi-party evaluation algorithm proceeds in the following steps:

Step 0: Features

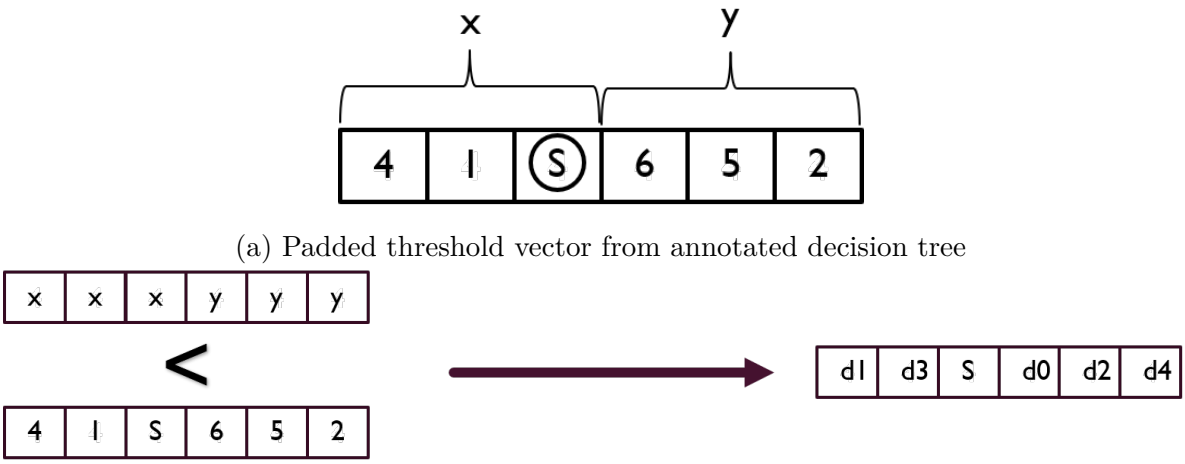
First, Maurice reveals the maximum multiplicity of any feature in a tree of the model to Sally, who then reveals it to Diane to enable the latter to set up an inference query. In the example in Figure 2.1, this is 3, which corresponds to the feature y, as it shows up in d_0, d_2, and d_4, whereas x only shows up in d_1 and d_3 (and hence has a multiplicity of 2). Diane then replicates each feature in her feature vector a number of times equal to this maximum multiplicity (for instance, yielding $[x, x, x, y, y, y]$), encrypts the replicated vector, and sends it to Sally.

Step 1: Comparison

Maurice uses the COPSE compiler to construct a vector containing all the thresholds in the tree, grouping together thresholds from decision nodes that use the same feature. This threshold vector is padded with a sentinel value S, as shown in Figure 4.2a, for when a feature has less than the maximum multiplicity, so that the threshold vector, representing the decision nodes in the forest, and Diane's feature vector are in one-to-one correspondence. Once this threshold vector is constructed, Maurice sends it to Sally. To perform inference, Sally pairwise compares the vector to Diane's encrypted feature vector to get the results of every threshold comparison, as illustrated in Figure 4.2b. Each element in the decision result vector is either a sentinel or corresponds to one of the decision nodes in the original tree. Note that this thresholding happens in a single parallel step regardless of the number of branches in the tree, as both Diane's features and the decision tree thresholds are packed into vectors.

Step 2: Reordering

Once Sally produces the decision vector, as in Figure 4.2b, she reorders all the branch decisions so they correspond to a pre-order walk of the tree, and removes the sentinel values.



(a) Padded threshold vector from annotated decision tree

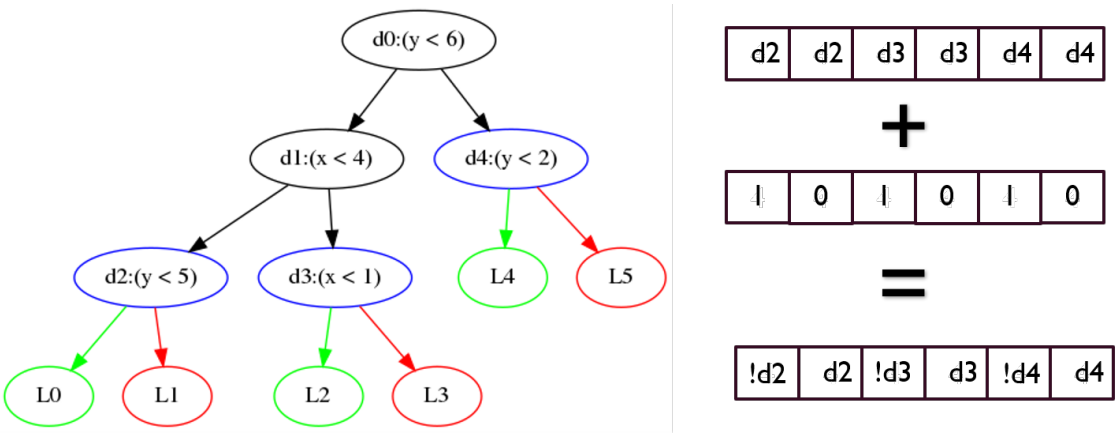
(b) Comparing replicated feature vector to padded threshold vector

Figure 4.2. Illustration of vectorized comparison step

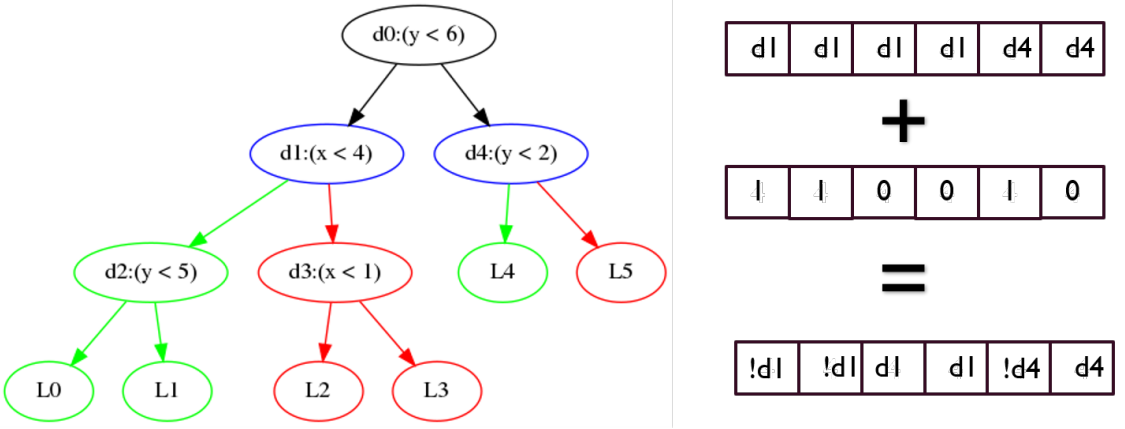
Step 3: Level Processing

Now that the decision results are in a canonical order, each level (counting up from the leaves) can be processed separately. At each level of the tree, Sally construct a binary mask encoding for each label whether it is downstream of the true or false branch at that level. Figure 4.3 shows how Sally processes levels 1 and 2. The nodes colored in blue are the nodes at that level, while the labels colored in green correspond to a 1 in the mask and are downstream of the "false" branch, and those in red correspond to a 0 in the mask and are downstream of the "true" branch. Note that d_4 is treated as part of level 1 *and* 2. This is because labels L_4 and L_5 are shallower than the other labels. Hence, d_4 is treated as though it exists at its own level as well as all lower levels.

Consider Level 1. Labels L_0, L_2, and L_4 correspond to the 'false' branch for these decisions, and labels L_1, L_3, and L_5 correspond to the 'true' branch. The decision results corresponding to that level are picked out of the vector from step 2 and XOR'ed with the mask to yield a boolean vector encoding whether each label could possibly be the classification result given the decision results at that level (in other words, for the i^{th} label to be possible, the i^{th} entry in the XOR'ed vector must be true).



(a) Processing at level 1



(b) Processing at level 2

Figure 4.3. Each level is processed individually

Step 4: Accumulation

Finally, once these vectors are collected for each level, Sally simply multiplies them all together to get a final label mask. As illustrated in Figure 4.4, an entry in this final vector is true if all the corresponding entries from the XOR'ed level vectors were true; this can only be true when the corresponding label is, in fact, the result returned by the decision tree. Note that the return value of the evaluation algorithm is not a single label but rather an N-hot bitvector with a single bit turned on for each tree. Section 4.3 discusses the reasoning behind this design choice, and Section 4.6.1 addresses the privacy implications.

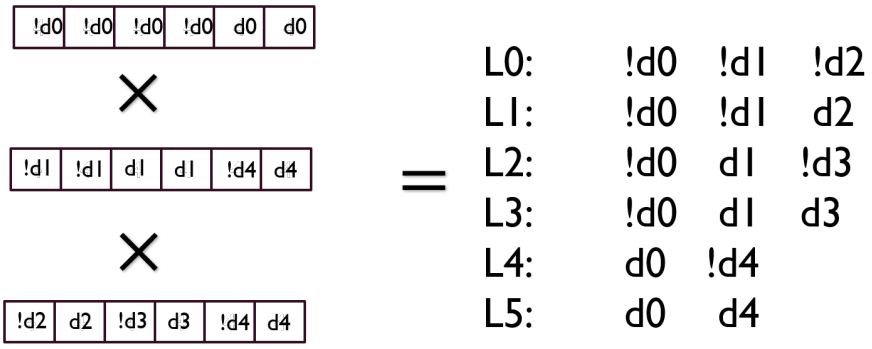


Figure 4.4. Level vectors are multiplied to yield the final result

A key point to notice about this algorithm is its inherent parallelizability, as each level can be processed entirely independently of the others. Another advantage is that all the computation at a given level can be packed into vectorized operations, as discussed in Section 2.3.1, which exposes even more algorithm-level parallelism. Finally, since all these steps are performed on encrypted data using FHE, nothing about the model is revealed to Diane or Sally aside from the maximum multiplicity, and nothing about the Diane's feature vector is revealed to anybody. Section 4.3 formalizes and describes in greater detail the exact primitives used to carry out this algorithm, and Section 4.6 informally discusses how configuring the FHE primitives in different ways yields different security properties.

4.2 Definitions & Preliminaries

4.2.1 Properties of Decision Trees

Decision Forest:

Consider a decision forest model M consisting of trees $T_1, \ldots, T_N$. Each tree T_i consists of a set of branches B_i (the interior nodes) and a sequence of labels L_i (the leaves). The labels in the sequence do not necessary have to be unique. We index all the branches in a tree by enumerating them in preorder; this indexing can be easily extended to the entire forest by not starting the count over for each new tree. The labels of the forest are similarly (separately) indexed.

All the data of a decision forest except for its branching structure is encoded in three vectors: $\mathbf{x}$, $\mathbf{f}$, and $\mathbf{t}$. Let $\mathbf{x} = (x_1, \ldots, x_n)$ be the set of features. Then $\mathbf{f}$ is the vector encoding which feature is compared against at each branch, and $\mathbf{t}$ is the threshold at each branch. For instance, if the branch B_7 has the condition $x_3 < 100$, then $f_7 = 3$ and $t_7 = 100$.

Properties of Nodes:

Each node in the tree has a *level*, which is the number of branches on the longest path from the node to a label (including itself; the level of a label node is 0), a *downstream set*, which is the set of all labels reachable from this node, and a *width* which is the size of the downstream set. An important consequence of these definitions is that, given a level d and a label L_{i}, there is a unique branch node B_{j} at level d that has L_{i} in its downstream set. To see why this is the case, consider two distinct nodes B_{j} and B_k that contain L_{i} in their downstream set. One of the two must be an ancestor of the other, since each node has a unique parent; thus, they cannot have the same level.

Properties of Models:

We define the *multiplicity* κ_{i} of a feature x_{i} to be the total number of times it appears in the model (in other words, κ_{i} is the number of times i appears in the vector $\mathbf{f}$). In the example tree in Figure 2.1, $\kappa_x = 2$ and $\kappa_y = 3$. The *maximum multiplicity* K of a forest is the maximum multiplicity of all its features (for the example tree, $K = 3$).

The *branching* b of a model is the total number of branch nodes it has; this is equivalent to the sum of the multiplicities for each feature, which in the example is $b = \kappa_x + \kappa_y = 2 + 3 = 5$. The *quantized branching* q is the product of the K and the total number of features; in other words, it is the branching if every feature had maximum multiplicity. In the example, since $K = 3$ and there are two features, $q = 6$.

4.2.2 Data Representation and Key Kernels

Representing Non-integral Values

Rather than try to securely perform bit operations on floating point numbers, we instead represent decision thresholds as fixed-point values with the precision p known at compile-time. A vector of k fixed-point values with precision p is represented with p bitvectors each of length k, with vector i holding the i^{th} bit of each element of the original vector. This peculiar "transposed" representation makes vectorizing computations easier later, allowing us to treat each bit independently while still performing comparisons in parallel.

Integer Comparison

We use the SecComp algorithm described by Aloufi, et. al [42]. Each "bit" of the values being compared is actually a bitvector packed as described above. The SecComp algorithm compares two equal-length bitstrings x and y lexicographically.

Matrix Representation

Matrices are represented as vectors of *generalized diagonals*. The i^{th} generalized diagonal d_{i} of an $m \times n$ matrix A is a vector defined as follows:

$$d_{\mathrm{i}} = (A_{0,\mathrm{i}}, A_{1,\mathrm{i}+1}, \ldots, A_{n-\mathrm{i},n}, A_{n-\mathrm{i}+1,0}, \ldots, A_{m,(m+\mathrm{i}) \bmod n})$$

Intuitively, this is the diagonal with an offset of i columns, wrapping around to the first column when necessary. For an $m \times n$ matrix there are always n generalized diagonals, each of which has length m.

Matrix Multiplication

The diagonal representation described above makes matrix/vector multiplication easier. To multipliy an $m \times n$ matrix M by an $n \times 1$ vector v, we use the algorithm described by Algorithms in HELib [39]. The i^{th} diagonal of M is multiplied component-wise by the vector

v rotated i slots. When $m \neq n$, the width of these two vectors will not be the same. If $m > n$, v is cyclically extended (e.g. $[x, y, z]$ becomes $[x, y, z, x, y, z, \ldots]$). If $n > m$, v is truncated after rotating. The vectors resulting from each such product are summed. This has the advantage of having a constant multiplicative depth of 1 regardless of the size of the matrix or vector.

Classification Result

The classification result is encoded into a bitvector with one slot for every label node in the forest. A slot in the bitvector holds a 1 if the corresponding label was the one chosen by its tree, and a 0 otherwise; in a forest with N trees, N slots in the bitvector will be set to 1.

Note that this approach to generating the results yields the classification decision of *each* component decision tree, rather than just the plurality classification. COPSE chooses the former approach as one point in the tradeoff space between efficiency and privacy, as discussed in Section 4.6.2.

4.3 The COPSE Algorithm

4.3.1 Algorithmic Primitives

Padded Threshold Vector

To carry out the comparisons in parallel, all the decision thresholds in the forest need to be packed into a single vector that is in one-to-one correspondence with Diane's feature vector. This packed threshold vector is actually a sequence of p bitvectors packed according to the description in Section 4.2.2, where p is the chosen fixedpoint precision (the i^{th} bitvector contains the i^{th} bit of each threshold). To prevent Diane from learning the exact structure of the decisions (i.e. which feature is thresholded against at each node of the forest), we group the thresholds in the vector by the feature they correspond to (so all the x_1's go at the beginning, followed by the x_2's, and so on). Revealing some information about how many times each feature is in the forest (in other words, κ_i) is, of course, unavoidable. We limit the scope of this information leak by only revealing the maximum multiplicity K of all the

features; for any feature with fewer than K occurences, the threshold vector is padded with some sentinel value S until its effective multiplicity is K. Our implementation chooses $S = 0$, but the exact value does not matter as the results from comparisons against a sentinel are removed later anyway.

Reshuffling Matrix

Once a boolean vector is produced containing the decision result for each node of the forest, it must be rearranged to correspond to the order of the branch enumeration. This also means removing the slots in the vector resulting from comparing against one of the sentinels used to pad the thresholds. In order to encode this reshuffling and sentinel removal, we construct a binary matrix $\mathcal{R}$ that, when multiplied by the decision result vector, produces a new vector with the results sorted correctly. The matrix $\mathcal{R}$ has a 1 in row i and column j if the j^{th} element of the padded threshold vector corresponds to the i^{th} branch of the decision tree. This means that there is exactly one of these in every row of $\mathcal{R}$, and at most one in every column, with the empty columns corresponding to the indices of the sentinel values.

Level Matrices

A level matrix is constructed for each level of the forest up to its maximum depth. For each label, the matrix at a given level selects the branch node above the label at that level. In the case where there is no such branch (for instance, there are branches above L_4 at level 1 and level 3, but none at level 2 in the example in Figure 2.1), the highest branch not exceeding that level is selected (this is d_4). The decision to do this is somewhat arbitrary; we could have just as easily chosen to use a higher level branch (such as d_0) instead, since what really matters is that every branch is represented in at least one of the levels. The level matrices are, like the reshuffling matrix, boolean matrices. A level matrix has a 1 in row i and column j if the branch node with index j is the one above the label node with index i at that particular level (or when no such branch exists, if it is the chosen replacement). Each row of the matrix has exactly one, and the number each column has is equal to the width of the corresponding branch.

Level Masks

For each level matrix there is a corresponding "mask," which is a boolean vector that encodes whether each label is on the "true" or "false" path from that level. For each label, we look at the corresponding branch above it (the same one determined by the level matrix). If the label is under the "true" path of that branch, we put a 0 in the corresponding slot of the mask vector; otherwise, we put a 1. Thus, given a vector of decision results for the branches above each label, XOR'ing this vector with the "mask" yields a new vector which has a 1 for any label that could be chosen by the decision result at that level. This means that multiplying (or AND'ing) together each of these vectors would result in a 1 only for the labels that each tree outputs.

4.3.2 Algorithm

The actual inference algorithm is implemented as vectorized computations using these structures. The overall flow is shown in Algorithm 3. First, the `SecComp` [42] primitive is applied to Diane's feature vector (Feats) and the padded threshold vector from Maurice's model (Thresh), which produces a boolean vector of decision results and sentinels. This vector is multiplied by the reshuffling matrix (Reshuf) using to produce a new boolean vector whose decision results correspond exactly to the branches of the forest in a preorder enumeration. For each level of the forest, the reshuffled vector is multiplied by the matrix for that level (Lvls) and then added to the mask for that level (Masks). Finally, every such vector is multiplied together to produce a single vector with a slot for each leaf node in the forest (Labels). This vector is sent back to Diane for decryption. By expressing the entire algorithm in terms of these vector operations and matrix multiplications, we are able to exploit a great degree of parallelism, and effectively scale the secure inference process to larger models.

Our algorithm uses Aloufi et al.'s `SecComp` [42] and Shoup's `MatMul` [39] as subroutines. The ability to express most of the computation in terms of MatMul is the key to the algorithm's vectorizability, since the MatMul routine is itself a set of parallel vector operations with constant multiplicative depth. Performing the computation for each level of the tree at once and then combining them all at the end lets us have a multiplication circuit that is only

logarithmic in the forest depth, instead of the naive approach with linear depth. Section 4.5 discusses the complexity and multiplicative depth of both the primitives and the algorithm as a whole in more detail.

Algorithm 3: Algorithm for vectorized inference

Input: Maurice: Thresh, Reshuf, Lvls, Masks
Input: Diane: Feats
Decisions ← `SecComp`(Thresh, Feats);
Branches ← `MatMul`(Reshuf, Decisions);
LvlResults ← ∅;
forall i ← 1 ***to*** *NumLevels* **do**
    LvlDecisions ← `MatMul`(Lvls[i], Decisions);
    LvlResults[i] ← LvlDecisions ⊕ Masks[i];
Labels ← `MultAll`(LvlResults);
Output: Labels

4.4 Compiler & Runtime

While the evaluation algorithm described above is effective at vectorizing the inference of a decision forest, it is not the most natural way in which such models are usually expressed. A compiler can solve this problem by taking a more natural representation of a trained model and automatically generating a program that creates these vectorizable structures and performs the inference algorithm. In this section, we discuss the implementation details of such a compiler.

Input Representation

The input to the compiler is a serialized trained decision forest model. The format consists of a line defining the label names as strings, followed by a line for each tree in the forest.

Each leaf node outputs the index of the label it corresponds to. For every branch node, the serialized output contains the index of its feature, the threshold value its compared to, and the serializations of its left and right subtrees respectively.

Compiler Architecture

COPSE is a staging metacompilation framework. The input to the first stage is a serialized decision forest model. The COPSE compiler translates this to a C++ program that uses the vectorizable data structures described in Section 4.3.1, specialized to the given model, and invokes the algorithmic primitives provided by the COPSE runtime. The generated C++ program is then compiled and linked against the COPSE runtime library to produce a binary which can be executed to perform secure inference queries.

Structuring COPSE as a staging compiler allows us to specialize the generated C++ code by (1) choosing an appropriate set of encryption parameters for the model being compiled and (2) selecting optimal implementations for the algorithmic primitives given the FHE protocol and implementation used by the runtime. In our sensitivity analysis in Section 4.7, we performed a sweep over the possible encryption parameters and found that for the models we were compiling, a single set dominated all the others. Since COPSE is currently targeted only to use the BGV implementation in HElib, a single set of optimal implementations is used for all the primitives. However, if COPSE were to use a different protocol and backend (for instance, SEAL and CKKS), these choices could matter and the staging compiler could appropriately tune the parameters and implementations.

COPSE Runtime

The runtime has datatypes that represent both plaintext and ciphertext vectors and matrices, as well as the parties playing the role of *model owner* (Maurice), *data owner* (Diane), and *evaluator* (Sally). It also exposes primitives to encrypt and decrypt models and feature vectors, and securely execute an inference query given an encrypted model and encrypted feature vector. The programmer can use these datatypes to encode their application logic, and then link against the generated C++ code to produce a binary that securely performs decision forest inference.

We use the HElib library [43] with the BGV protocol [44] as our framework for homomorphic encryption. This library provides low-level primitives for encrypting and decrypting plaintext and ciphertexts, homomorphically adding and multiplying ciphertexts, and gen-

erating public/secret key pairs. HElib also supports ciphertext packing which gives us the vectorizing capabilities we need.

4.5 Complexity Analysis

This section characterizes the complexity of COPSE. This complexity is parameterized on various parameters of the decision forest model: the number of branches b, the total number of levels d, the fixedpoint precision p, and the quantized width q. (Definitions of these parameters can be found in Section 4.2.1.) The complexity of FHE circuits is characterized by two elements: (1) the number of each kind of primitive FHE operation and (2) the *multiplicative depth* of the FHE circuit. The former captures the "work" needed to execute the circuit. The latter, characterized by the longest dependence chain of multiplications in the circuit, determines the encryption parameters needed to evaluate the circuit accurately (higher multiplicative depth requires more expensive encryption, or bootstrapping).

The FHE operations used to express the amount of work are: (1) *Encrypt*, which produces a single ciphertext from a plaintext bitvector; (2) *Rotate*, which rotates all the entries in a vector by a constant number of slots; (3) *Add*, which computes the XOR of two encrypted bitvectors; (4) *Multiply*, which computes the AND of two encrypted bitvectors, and (5) *Constant Add*, which computes the XOR of an encrypted bitvector with a plaintext one. The *Multiply* operation incurs a multiplicative depth of 1, and all the rest incur a multiplicative depth of 0.

Table 4.1 characterizes the steps of the COPSE algorithm, in terms of the number of FHE operations and their multiplicative depth, as well as the cost of encrypting the data and models (which do not factor in to multiplicative depth, as they are separate from the circuit). Table 4.2 shows the overall cost of COPSE, including combining the multiplicative depths of the individual steps according to their dependences in the overall circuit. Note that the cost of processing a single level is incurred d times, but the level processing steps occur in parallel in the FHE circuit, so altogether the level processing only contributes 1 to the multiplicative depth.

Table 4.1. Operation counts and multiplicative depth for COPSE

(a) Complexity for Secure Comparison

Operation	Number of Ops
Add	$4p - 2$
Constant Add	p
Multiply	$p \log p + 3p - 2$

Multiplicative depth: $2 \log p + 1$

(b) Complexity for processing a single level (repeats d times)

Operation	Number of Ops
Rotate	b
Add	$b + 1$
Multiply	b

Multiplicative depth: 1

(c) Complexity for accumulating results from all levels

Operation	Number of Ops
Multiply	$2d - 2$

Multiplicative Depth: $\log d$

(d) Complexity for encrypting model

Operation	Number of Ops
Encrypt	$p + q + d(b + 1)$

(e) Complexity for encrypting data

Operation	Number of Ops
Encrypt	1

Table 4.2. Total Evaluation Complexity

Operation	Number of Ops
Encrypt	$1 + p + q + d(b + 1)$
Rotate	$q + db$
Add	$4p - 2 + q + d(b + 1)$
Constant Add	p
Multiply	$p \log p + 3p + q + db + 2d - 4$

Multiplicative Depth: $2 \log p + \log d + 2$

Table 4.3. Data revealed to each notional party in two-party configurations

Scenario	Revealed to S	Revealed to M	Revealed to D
$S, M = D$	q, b, d	$\emptyset$	$\emptyset$
$S = M, D$	$\emptyset$	$\emptyset$	K, b
$S = D, M$	q, b, K, d	$\emptyset$	q, b, K

Table 4.4. Data revealed to each party in three-party configurations

Scenario	Revealed to S	Revealed to M	Revealed to D
S, M, D, no collusion	q, b, d, K	$\emptyset$	K, b
S, M, D, S colludes with M	everything	everything	K, b
S, M, D, S colludes with D	everything	$\emptyset$	everything

4.6 Security Properties

This section describes the various security properties of decision forest programs built using COPSE. Section 4.6.1 discusses information leakage between the parties, while Section 4.6.2 discusses the privacy implications of different *design* decisions in COPSE.

4.6.1 Information Leakage

COPSE has three *notional* parties: the model owner **M**aurice, the data owner **D**iane, and the server **S**ally. Maurice owns τ, $\mathcal{L}$, $\mathcal{R}$, m, q, b, and K, while Diane owns the feature vector f. Sally owns nothing.

Two Physical Parties

FHE is inherently a two-party protocol, so although the secure inference problem has three notional parties, our system focuses on the cases where there are only two *physical* parties (i.e., two of the notional parties are actually the same person). There are three scenarios:

1. Where $M = D$; for instance, if the model and data are owned by the same party, which offloads the inference to an untrusted server. This is the standard "computation offloading" model used by most FHE applications [9, 10, 18].

2. Where $M = S$; if the model is stored on some server which allows clients to send encrypted data for classification

3. Where $D = S$; if the model is trained and sent directly to a client for inference, but the client must be prevented from reverse-engineering the model.

In Table 4.3 we describe what data is explicitly revealed or implicitly leaked to each party. When $M = D$, obviously neither party can leak information to the other. However, because matrices are encrypted as a vector of ciphertexts with one per column (diagonal), S learns the number of columns in each matrix. This translates to learning the number of branches b from each level matrix $\mathcal{L}$, and learning the quantized width q from the reshaping matrix $\mathcal{R}$. Furthermore, since the level masks and matrices are stored separately, S also learns the maximum depth of the forest.

When $S = M$, neither S nor M can leak information to each other. However, M must explicitly send the value of K to D to get feature vectors with the right padding. When the inference result is sent back, D also learns $b + 1$, as it is the length of the final inference vector.

When $S = D$, once again neither S nor D leak information to each other. However, this time M not only reveals K and b to both S and D the same way as in case (2), but q is also leaked through the widths of the matrices, as well as d.

Three Parties

When S, M, and D are separate physical parties that do not collude, M necessarily leaks to S the values of b, q, and d, as well as revealing K. S then reveals K to D, and M leaks b to D. Even though M and D use the same key pair, because neither colludes with S, neither ever gets access to the other's ciphertexts, and privacy between the two is therefore preserved. However, if one of the parties does collude with S, they gain access to the other party's ciperhtexts which can then be easily decrypted. Thus in the case where there is collusion between M or D and S, everything is leaked. Table 4.4 summarizes these results.

Since it is difficult to convince both M and D that the other is not colluding with S, we see that attempting to run this protocol with three physical parties using single-key FHE is unreasonable. There has been a lot of prior work on multikey FHE schemes [29, 30] and threshold FHE, which uses secret sharing to extend single-key FHE to work in a multiparty setting [28]. These schemes act as "wrappers" that construct a new, joint key pair for FHE (in this case shared by D and M), and hence can be applied directly to COPSE at the cost of introducing additional rounds of communication and additional encryption/decryption steps.

4.6.2 Security Implications of COPSE design

The design of COPSE admits different points in the design space that trade off security and performance. Here, we discuss the implications of the design points that we chose.

Feature Padding

Choosing to have Diane replicate and pad her feature vector is a tradeoff we make between the performance and security of COPSE. To avoid requiring Diane to do any preprocessing beyond replicating her feature vector, we would need to explicitly reveal the multiplicity of each feature used in the model. By requiring the feature vector to be padded, we only reveal the maximum feature multiplicity of the model must be explicitly revealed.

We could even avoid revealing the exact maximum multiplicity, and instead only reveal an upper bound, simply by adding several extra sentinel values to each feature in the threshold vector. The performance overhead of this would be minimal, except a slightly more expensive matrix multiply to remove the extra sentinel values (the size of this overhead scales with how loose the given upper bound is).

To avoid leaking *any* multiplicity information, we could also relax the requirement that the Diane replicate her features at all, instead accepting a vector that lists each feature once, and requiring that the server carry out the necessary replication directly on the ciphertext vector. While this does prevent Diane from learning anything about feature multiplicities in the mode, it has the effect of replacing several (cheap) plaintext replication operations with their equivalent ciphertext ones, which are much more expensive.

Returning Classification Bitvectors

Returning the bitvector of classification results rather than accumulating them to return a single label leaks some information about the structure of the model to Diane.

First, it requires a "codebook" (i.e. a map from each position in the bitvector to the label it represents) to be revealed to Diane. This reveals the order of the labels in the constituent trees of the forest (though not the "boundaries" between the trees). It is possible to avoid leaking the order of the label nodes by first having the server generate a random permutation to apply to the decision result bitvector (via a plaintext matrix/ciphertext vector multiplication), then applying the same permutation to the codebook.

Shuffling the codebook still reveals information about the model structure; in particular, it leaks how many leaf nodes correspond to each label. For instance, knowing whether a particular label is output by most of the leaves in the forest versus only being output by a single leaf potentially reveals something about how "likely" that label is to be chosen. This can also be avoided if the server pads both the codebook and the classification result bitvector with random extra labels before returning both of them; this step can be folded into the shuffling step as well, so it has minimal extra cost.

COPSE currently assumes that the codebook is already known to Diane, and performs neither shuffling nor padding.

Another data leak is the label result chosen by each tree. In other words, Diane learns that, for instance, two trees chose X and three chose Y, rather than simply learning that the final classification is Y. This is an unavoidable consequence of COPSE's design of returning a bitvector of results rather than performing the reduction server-side. The effect of this design is to put the burden of accumulating all the chosen labels into a single classification on Diane. Doing so necessarily requires revealing all the chosen labels.

Accumulation could be done by Sally to avoid this leak, at the cost of expensive ciphertext operations, including potentially multiple interactive rounds to change the plaintext modulus and count up the occurrences of each label. Thus, while this design is somewhat more secure, it adds *communication complexity* in addition to computational complexity.

4.7 Evaluating COPSE

We evaluate COPSE in several ways. First, we evaluate how well COPSE performs against the prior state-of-the-art in secure decision forest inference, Aloufi, et. al [42].. This evaluation looks at sequential and parallel performance, and focuses on the classic, offloading-focused privacy model where the model and data are owned by one party, and the server is another party (see Section 4.6). Second, we consider COPSE's ability to handle different party configurations, in particular, when the server and model are owned by one party, and the data by another. Finally, we use microbenchmarks to understand COPSE's sensitivity to different aspects of the models: depth, number of branches, and feature precision.

4.7.1 Benchmarks, Configurations, and Systems

To evaluate COPSE, we synthesized several microbenchmark models that varied the number of levels, number of branches, and the bits of precision used for expressing thresholds. We use these microbenchmarks both for performance studies (this section) and for sensitivity studies (Section 4.4). In addition to microbenchmarks, we obtained open-source ML data sets to train decision forests for real-world benchmarks, `income` [45] and `soccer` [46]. We

used the scikit-learn library [47] to train random forest classifiers on these data sets. For each of the real-world data sets, we generated two differently-sized models (suffixed `5` and `15`), reflecting the number of decision trees comprising each forest.

Configuring HELib involves setting several encryption parameters: the *security parameter*, the number of *bits* in the modulus chain, and the number of *columns* in the key-switching matrices. Increasing the security parameter results in larger ciphertexts, increasing security and computation time; increasing the number of bits in the modulus chain increases the maximum multiplicative depth the circuit can reach; and changing the number of columns in the key-switching matrices affects the available vector widths. We performed a sweep over the range of possible encryption parameters for our models, and found a single set of parameters that worked sufficiently well. Table 4.5 lists the encryption parameters we used. (Note that it is possible that for other models, or other FHE implementations, other parameters will be superior; autotuning these parameters can be incorporated into the staging process, as described in Section 4.4.)

All experiments were performed on a 32-core, 2.7 GHz Intel Xeon E5-4650 server with 192 GB of RAM. Each core has 256 KB of L2 cache, and each set of 8 cores shares a 20 MB last-level-cache.

Table 4.5. Optimal encryption parameter values

Parameter	Value
Security Parameter	128
Bits	400
Columns	3

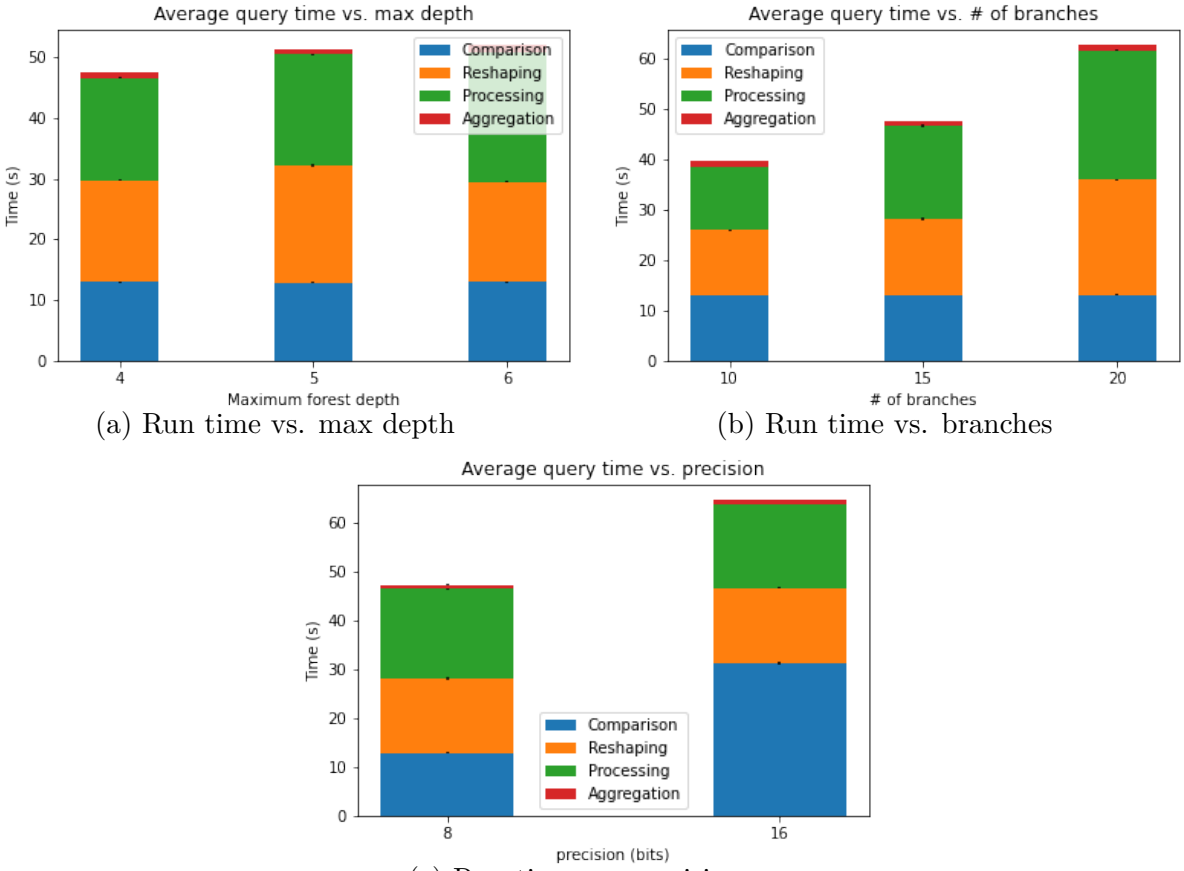
4.7.2 COPSE Performance

Our first evaluation focuses on the sequential and parallel performance of COPSE. Our baseline for COPSE is the state-of-the-art approach for performing secure decision forest evaluation in FHE, by Aloufi, et. al [42]. Because Aloufi et al.'s implementation was not available, we implemented their algorithms ourselves. We made our best effort to optimize our reimplementation, including introducing parallelism with Intel's Thread Building

Blocks [48] (indeed, our implementation appears to scale better than Aloufi et al.'s reported scalability). Crucially, both our baseline and COPSE use the same FHE library, and the same implementation of SecComp, which was introduced by Aloufi, et. al [42]..

We evaluated both implementations on two primary criteria: how quickly the compiled models could execute inference queries, and how effectively COPSE was able to take advantage of parallelism to scale to larger models. For each model, we performed 27 inference queries, in both single-threaded and multithreaded mode. We report the median running time across these queries (confidence intervals in all cases were negligible).

Figure 4.6 shows the relative speedups over prior work for each model compiled using COPSE. We see that we have a substantial speedup over the baseline, ranging from 5× to over 7×, with a geometric mean of close to 6×.



(a) Run time vs. max depth

(b) Run time vs. branches

(c) Run time vs. precision

Figure 4.5. Run time of microbenchmarks

Multithreading

Next, we ran inference queries on the models with multithreading enabled. For all queries, we ran the systems using 32 threads. While the individual queries were mutithreaded, they were still executed sequentially one after another.

Figure 4.7 shows the multithreaded speedup of COPSE over single-threaded COPSE. Note that in this study, we evaluated even larger models for our real-world datasets, as COPSE is able to evaluate these in a reasonable amount of time while our baseline is not. We see that while parallel speedup for the microbenchmarks is relatively modest (around 2.5×), parallel speedup for the real-world models is much better (almost 5×), as we might expect: the real-world models are larger, and present more parallel work.

Figure 4.8 compares COPSE's parallel performance to our baseline. We note that it appears that COPSE scales worse than the baseline (if they scaled equally well, the speedups in Figure 4.8 would match the speedups in Figure 4.6). The source of this seeming shortcoming is subtle. COPSE's design makes heavy use of ciphertext packing to amortize the overheads of FHE computation. This ciphertext packing essentially consumes some of the parallel work in the form of "vectorized" operations, even in the single-threaded case, leaving the COPSE implementations with less parallelism to exploit using "normal" parallelization techniques. In contrast, all of these parallelism opportunities can *only* be exploited by multithreading in the baseline (and recall that our baseline implementation appears to scale better than the original implementation). We can observe this effect by noting that the gap in scaling is smaller for the larger, real-world models (`income5` and `soccer5`): there is more parallelism to start with, so even after ciphertext packing, there is more parallelism for COPSE to exploit.

4.7.3 Different Party Setups

As discussed in Section 4.6, the two-party setup used by COPSE admits different configurations for the identities of these parties. We would expect to see speedup if Maurice and Sally are the same party (so the models can be represented in plaintext) compared to when Maurice and Diane are the same party (o the model has to be encrypted). Figure 4.9 shows the speedup of inference when using the second configuration (plaintext models) versus the

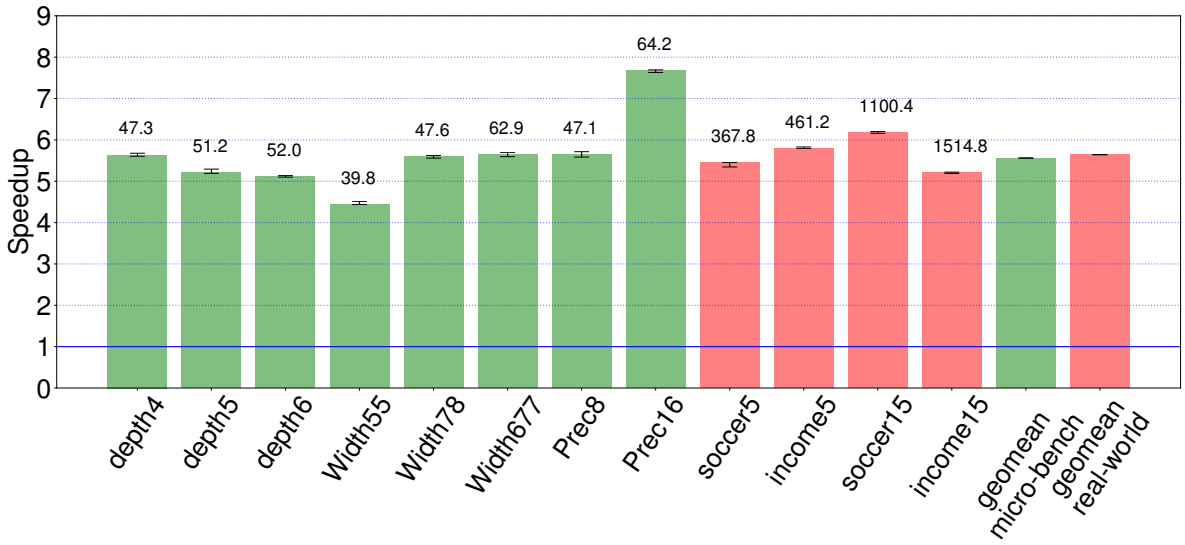


Figure 4.6. Speedup of COPSE-compiled models over our implementation of Aloufi, et. al [42] when both are single-threaded. The number on top of each bar is the median running time (in milliseconds) for that model using COPSE.

Table 4.6. Microbenchmark specifications

Model name	Max. depth	Precision	# of trees	# q of branches
depth4	4	8	2	15
depth5	5	8	2	15
depth6	6	8	2	15
width55	5	8	2	10
width78	5	8	2	15
width677	5	8	3	20
prec8	5	8	2	15
prec16	5	16	2	15

first (ciphertext models). As expected, we see that plaintext models result in substantial speedups of roughly 1.4×.

4.7.4 Evaluation on Microbenchmarks

To better understand the different components of COPSE, we used eight randomly-generated forests with different properties: feature precision, maximum levels, number of trees, and number of branches. The details on the size of each forest can be found in Table 4.6. Every forest had 2 features and 3 distinct labels. Figure 4.5 shows the median running time

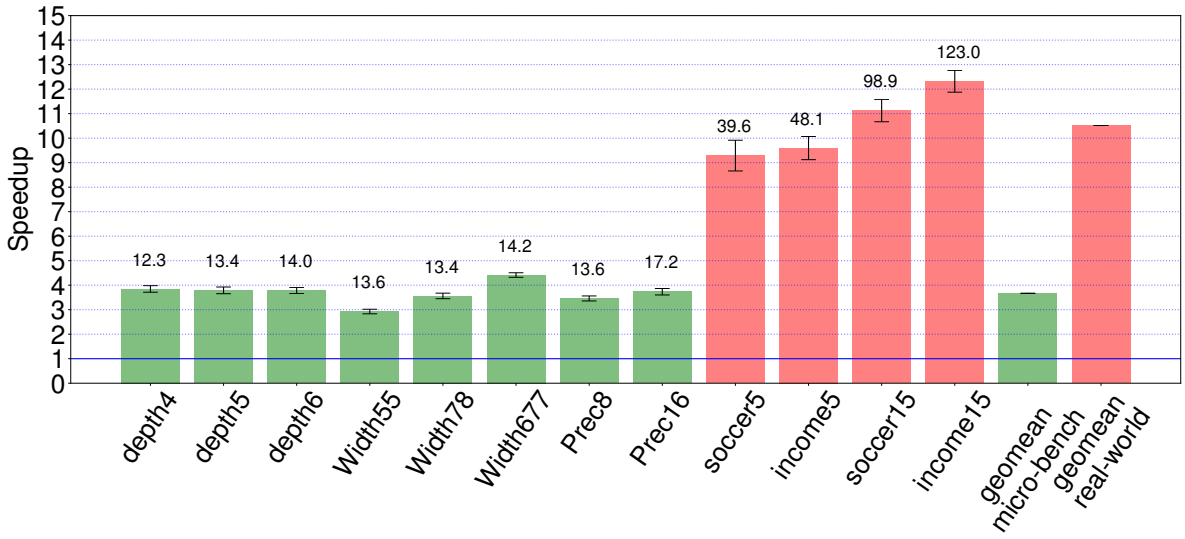


Figure 4.7. Speedup that COPSE-compiled models experience when multi-threaded instead of single-threaded. The number on top of each bar is the median run-time (in milliseconds) for multithreaded inference.

for each model, broken down by the time each step took (comparison, reshuffling, processing levels, and aggregating). To facilitate comparison, each sub-figure compares models that are similar except for a particular parameter under test.

Effects of depth

Figure 4.5a shows models that differ in terms of tree depth. Comparison and reshaping times are largely unaffected by the maximum forest depth, whereas the total level processing time increases approximately linearly. Aggregation time is logarithmic in depth, but this is negligibly small compared to the rest of the evaluation. This makes sense because at each depth level there is approximately an equal amount of work to be done.

Effects of branching

Figure 4.5b shows models that differ in terms of number of branches. While the comparison time is unaffected by branching, both the reshaping time and total level processing time are. The relationship between branching and reshaping is close to linear, as reshaping

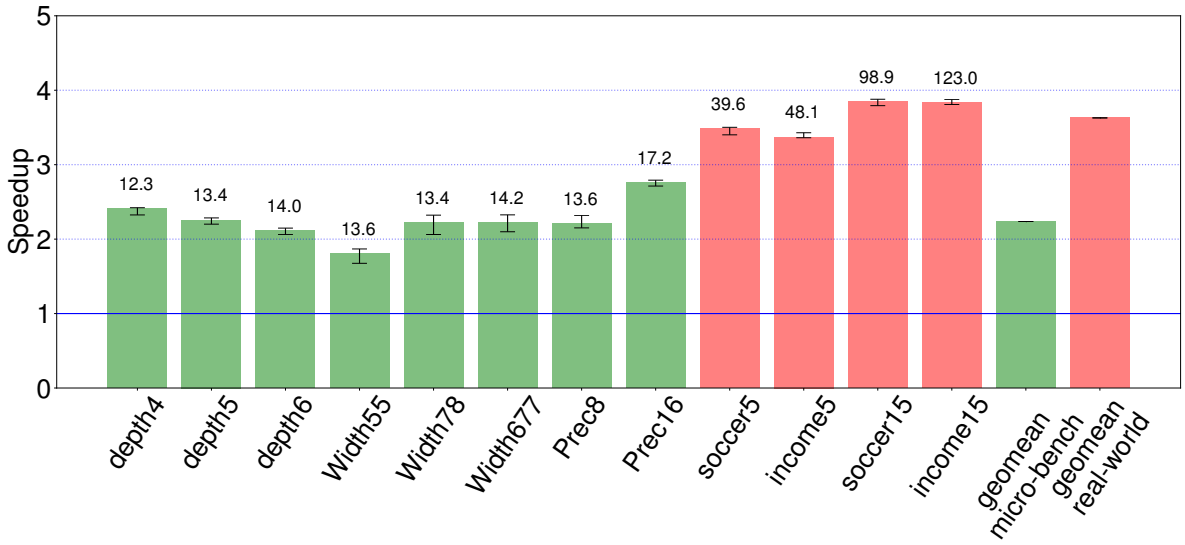


Figure 4.8. Speedup of COPSE-compiled models over our implementation of Aloufi, et. al [42]. when both are multithreaded. The number on top of each bar is the median run-time (in milliseconds) for multithreaded COPSE.

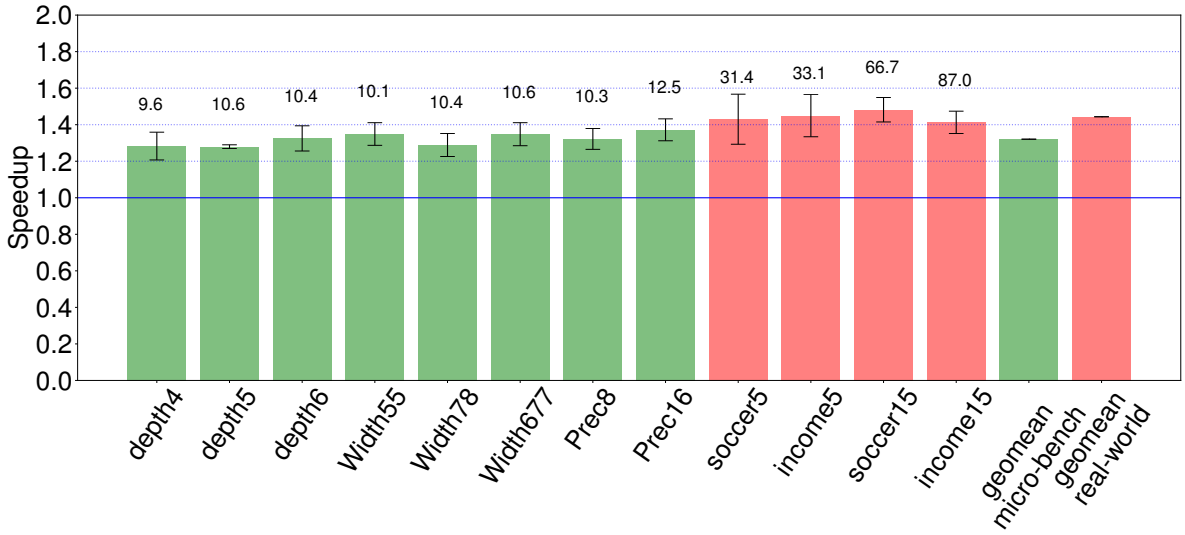


Figure 4.9. Speedup of inference queries executed on plaintext models (when Maurice = Sally) compared to encrypted models (when Diane = Maurice). The number on top of each bar show the median inference run-time (in milliseconds) on the plaintext models.

actually depends linearly on the quantized branching. By contrast, the total level processing time is directly proportional to the number of branches. This is because the matrix at each level has a number of columns equal to the number of branches. Thus for a model with twice

as many branches, the corresponding matrices will be twice as wide and take twice as long to process without multithreading.

Effects of precision

Finally, Figure 4.5c shows models that differ in terms of feature precision. Reshaping, level processing, and aggregation are, as expected, unaffected by changing the model precision. However, as suggested by the complexity analysis in Section 4.5, comparison time increases super-linearly with precision. This superlinear relationship is evident from the data in Figure 4.5c.

5. COIL

> To die, to sleep–
> To sleep–perchance to dream: ay, there's the rub,
> For in that sleep of death what dreams may come
> When we have shuffled off this mortal coil,
> Must give us pause.
>
> William Shakespeare

The previous two chapters have explored compiler cryptosystem co-design from two different angles. In Chapter 3, we used Coyote to show how to adapt compilation techniques like SLP vectorization to FHE programs; i.e., "compilers for crypto". In Chapter 4, we demonstrated how using the correct abstractions at the *cryptographic level* enables us to build even more productive compilers; i.e., "crypto for compilers". In this chapter, we finally unify these two perspectives, in a rather literal way, with COIL: a compiler for homomorphic circuits with control flow.

Oblivious Control Flow

We begin the discussion by recalling an observation made in the previous chapter: We can often improve upon an unstructured linearization strategy like multiplexing by taking advantage of some structure present in the control flow of the source program. While COPSE does this to exploit the vectorizability of the decision tree structure, in this chapter we extend these ideas to apply to a more general class of programs by describing an alternative to the usual "muxing" strategy. We introduce an intermediate representation (IR) called *path forests* (Section 5.2). 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 cannot happen) and *specialization* (instantiating ciphertexts whose values

are constrained along each path). Pruning and specialization can be thought of as versions of dead code elimination and constant propagation that take into account path-dependent information. These are particularly effective in the FHE setting because of the performance benefits of removing ciphertext computation (Section 5.5.3 discusses this further).

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[1]:

```
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[2] 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

[1]↑The remainder of this chapter adopts the convention of underlining ciphertext values in all code snippets.
[2]↑For ease of presentation, the path forests in the examples are written differently from the formal grammar defined in Figure 5.5

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⟩)
```

Representing the program as a path forest has another important consequence: Because all of the branching conditions are pulled out to the top, we can now represent the path forest as a *decision tree* (Section 5.2.6). In particular, this means that instead of resorting to evaluating the final forest with a mux network as above, we can now appeal to the more sophisticated decision tree inference algorithm we developed in the previous chapter! In fact, as we discuss in Section 5.4.4, the ability to use COPSE here contributes substantially to our performance improvements.

5.1 COIL Overview

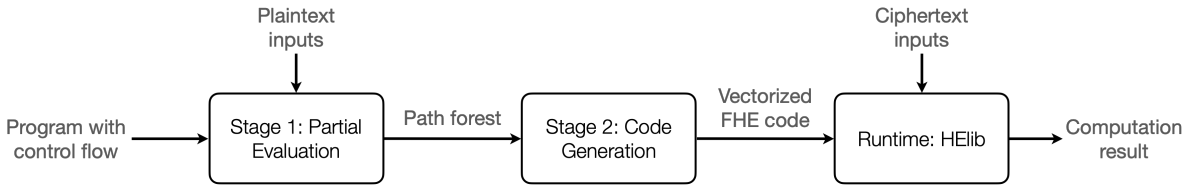


Figure 5.1. COIL pipeline

Our goal is a compiler for FHE programs that contain oblivious control flow. Naïvely compiling control flow into muxes 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.

```
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, 2), ptxts(3, 5)⟩ in
let key = ctxt(0) in
let idx = index(keys, key, 0) in
lookup(values, idx, 0)
```

Figure 5.2. 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 5.1). 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 5.6).

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 [49].

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 5.2, 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[3], returning `-1` if the key is not found.

5.1.1 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.

[3]↑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 5.2.1).

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 5.2 yields a forest with a single path for each possible value of the key (Figure 5.3b). Lets see why:

Assume the plaintext arrays returned on line 23 are `[1; 2; 3]` and `[1; 4; 9]` for `keys` and `values`, respectively. First, call to the `index` function on line 25 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 5.3a.

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 5.3.1).

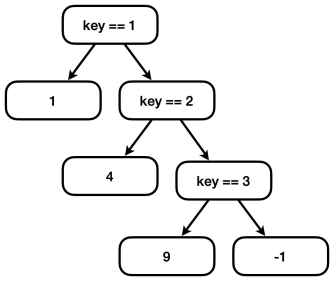
In a final round of pruning and specialization, the paths that require conflicting values of `idx`, and are thus infeasible, are removed, and the now-plaintext indices into the `values` array are resolved, reducing Figure 5.3a into the final forest in Figure 5.3b.

```
[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

Figure 5.3. Applying the path forest evaluation technique to Figure 5.2

5.1.2 COIL Program = Computation + Decision Tree

Looking at the path forest in Figure 5.3b, 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 flow can therefore be encoded as a *decision tree* (as in Figure 5.3c), 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.

Thus, COIL's transformations allow the program to be split into two phases: computing the labels, and then evaluating the decision tree to select the correct label. Because the labels, and the branching conditions themselves, are branch-free code, we can readily exploit the ciphertext-packing capabilities of many FHE schemes (Section 2.3.1) and off-the-shelf FHE vectorization techniques [20, 26, 50] to parallelize these computations.

All that remains, then, is to generate code for efficient decision tree inference, which we do using the COPSE algorithm [49]. COPSE exploits the ciphertext-packing capabilities of many FHE schemes 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 5.3.1.

Section 5.2.6 discusses our code generation strategy in more detail.

5.1.3 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[4] involves first computing a "one-hot" indicator vector encoding the position of the key:

```
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 5.2. 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,

[4]↑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 [51, 52]

COPSE's parallelized level evaluation and accumulation (described in more detail in Section 5.3.1) 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 5.4.2.

5.2 Language Syntax & Semantics

This section first describes the COIL language (Section 5.2.1), then explores in more detail the transformations done in the first compilation stage (Sections 5.2.2-5.2.3), and finally discusses how COIL generates code from a path forest.

NumExpr	e	::=	x	variable
		|	n	numeric literal
		|	**ptxt**(n)	plaintext input
		|	**ctxt**(n)	ciphertext input
		|	**if** (b) {e_1} **else** {e_2}	conditional
		|	**let** $x = p_1$ **in** p_2	variable definition
		|	**mux**(b, e_1, e_2)	multiplexer
		|	$arr[\mathrm{e}_1]$	array index
		|	**update** arr { $n_1 \rightarrow \mathrm{e}_1, \ldots, n_k \rightarrow \mathrm{e}_k$ } **in** p	array update
		|	($\mathrm{e}_1 \cdot \mathrm{e}_2$)	arithmetic
		|	f ($\mathrm{e}_1, \ldots, \mathrm{e}_n$)	function call
BoolExpr	b	::=	$\mathrm{e}_1 \equiv \mathrm{e}_2$	equality
		|	$\mathrm{e}_1 < \mathrm{e}_2$	inequality
		|	$\neg b$	negation
CoilProgram	p	::=	e	numeric expression
		|	[e_1; e_2; ...; e_n]	array literal
		|	**ctxts**(n_1, n_2)	ciphertext array
		|	**ptxts**(n_1, n_2)	plaintext array
		|	**let** $f = \lambda(x_1, \ldots, x_n) \Rightarrow$ e **in** p	function definition

Figure 5.4. Syntax of the COIL language. Note that the **mux** production is only added for use in Section 5.2.5, and is not directly used in any of our benchmarks.

Expr	e	::=	n	numeric literal
		|	$\mathbf{ptxt}(n)$	plaintext literal
		|	$\mathbf{ctxt}(n)$	ciphertext literal
		|	$\mathrm{e}_1 \cdot \mathrm{e}_2$	arithmetic
		|	$\ldots$	
PathClause	c	::=	$\mathrm{e}_1 \equiv \mathrm{e}_2$	equality
		|	$\mathrm{e}_1 < \mathrm{e}_2$	inequality
		|	$\neg b$	negation
Path	p	::=	$[c_1] \ldots [c_n]$	
PathForest	f	::=	$(p_1, \mathrm{e}_1); \ldots; (p_n, \mathrm{e}_n)$	

Figure 5.5. 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.

$$
\begin{aligned}
[\![(p, n)]\!]^{\Gamma}_{U} &\triangleq (p, n) && \text{where } n \text{ is a } \mathbf{ptxt}, \mathbf{ctxt}, \text{ or literal} \\
[\![(p, x)]\!]^{\Gamma}_{U} &\triangleq (p, \Gamma(x)) && \text{if } x \text{ is a variable bound in } \Gamma \\
[\![(p_1, \mathrm{e}_1); \ldots; (p_n, \mathrm{e}_n)]\!]^{\Gamma}_{U} &\triangleq [\![(p_1, \mathrm{e}_1)]\!]^{\Gamma}_{U}; \ldots; [\![(p_n, \mathrm{e}_n)]\!]^{\Gamma}_{U} && \\
[\![(p, \mathbf{let}\ x = \mathrm{e}_1\ \mathbf{in}\ \mathrm{e}_2)]\!]^{\Gamma}_{U} &\triangleq [\![(p', \mathrm{e}_2)]\!]^{\Gamma[x \mapsto \mathrm{e}']}_{U}; \ldots && \text{for each } (p', \mathrm{e}') \in [\![(p, \mathrm{e}_1)]\!]^{\Gamma}_{U} \\
[\![(p, \mathbf{if}\ (b)\ \{\mathrm{e}_1\}\ \mathbf{else}\ \{\mathrm{e}_2\})]\!]^{\Gamma}_{U} &\triangleq [\![(p'\ [b'], \mathrm{e}_1); (p'\ [\neg b'], \mathrm{e}_2)]\!]^{\Gamma}_{U}; \ldots && \text{for each } (p', b') \in [\![(p, b)]\!]^{\Gamma}_{U} \\
[\![(p, f(\mathrm{e}_1, \ldots, \mathrm{e}_n))]\!]^{\Gamma}_{U} &\triangleq [\![(p\ p'_1 \ldots p'_n, \mathrm{e})]\!]^{\Gamma[x_\mathrm{i} \mapsto \mathrm{e}'_\mathrm{i}]}_{U}; \ldots && \text{where } \Gamma(f) = \lambda(x_1, \ldots, x_n) \Rightarrow \{\mathrm{e}\}, \\
& && \text{for each } (p'_\mathrm{i}, \mathrm{e}'_\mathrm{i}) \in [\![(p, \mathrm{e}_\mathrm{i})]\!]^{\Gamma}_{U} \\
[\![(p, \mathrm{e}_1 \cdot \mathrm{e}_2)]\!]^{\Gamma}_{U} &\triangleq [\![(p'_\mathrm{i}\ p''_\mathrm{j}, \mathrm{e}'_\mathrm{i} \cdot \mathrm{e}''_\mathrm{j})]\!]^{\Gamma}_{U}; \ldots && \text{for each } (p'_\mathrm{i}, \mathrm{e}'_\mathrm{i}) \in [\![(p, \mathrm{e}_1)]\!]^{\Gamma}_{U} \\
& && \text{and } (p''_\mathrm{j}, \mathrm{e}''_\mathrm{j}) \in [\![(p, \mathrm{e}_2)]\!]^{\Gamma}_{U}
\end{aligned}
$$

Figure 5.6. The transformations done by the COIL compiler are implemented in the $[\![\cdot]\!]^{\Gamma}_{U}$ 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.

5.2.1 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 5.4). COIL uses a *staging compiler*, which means that programs are compiled in multiple *stages* [53]. The first stage compiles a COIL program down to the *path forest IR* (Figure 5.5) by *partially evaluating* it with respect to the public inputs when they become available, and then performs optimizations on this IR (Section 5.2.4) [54]. 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[5]
- Recursion must terminate *based on publicly known inputs*, since otherwise, evaluating a recursive function would leak something about its termination condition (Section 5.2.3).

In the example program in Figure 5.2, 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 (`cur` is a plaintext and `array` has statically known size), the unfolding can be done entirely in the first stage.

5.2.2 Compilation

The first stage of compilation requires lowering a COIL program (Figure 5.4) to the path forest intermediate representation, the syntax for which is shown in Figure 5.5, 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.

[5]↑Ciphertext indices are possible by, for example, writing an array indexing function (such as the one described in Section 5.1.3). 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 ...
```

Figure 5.7. 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.

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 5.6 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 $\mathrm{e}_1 \cdot \mathrm{e}_2$, 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** $x = \mathrm{e}_1$ **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 5.2.3). This exponential effect is discussed further in Section 5.5.

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:

$$(x < y, z + 2 * y); (x \geq y, z + 2 * x)$$

5.2.3 Optimizations on Path Forests

Recall from Section 5.1 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.

Specialization

Consider again the example in Section 5.2.2. 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:

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

(Here, the **eval** function implements the expected 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 5.7 in which the programmer writes a function that loops to compute the midpoint of two indices[6]. 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.

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 5.6 with the following:

$$[\![(p, \mathbf{if}\ (b)\ \{\mathrm{e}_1\}\ \mathbf{else}\ \{\mathrm{e}_2\})]\!]^{\Gamma}_{U} \triangleq \begin{cases} [\![(p\ p', \mathrm{e}_1)]\!]^{\Gamma}_{U}; \ldots & p \implies b' \\ [\![(p\ p', \mathrm{e}_2)]\!]^{\Gamma}_{U}; \ldots & p \implies \neg b' \\ [\![(p'\ [b'], \mathrm{e}_1); (p'\ [\neg b'], \mathrm{e}_2)]\!]^{\Gamma}_{U}; \ldots & \text{otherwise} \end{cases}$$

[6]↑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.

When encountering a branching condition b, we generate the queries $p \wedge b$ and $p \wedge \neg b$ and discharge them to a solver[7]; 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!

5.2.4 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 5.6 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 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 5.2.1 (that recursion termination conditions depend only on plaintexts) 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)
```

[7]↑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.

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:

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

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

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

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.

5.2.5 Re-Folding

Recall that the transformation described in Figure 5.6 produces a forest with a number of paths exponential in the number of branches in the original program. In this section, we describe a transformation that takes a path forest in normal-form and "folds together" adjacent paths that contain repeated computations. The core of the transformation is implemented in the following rewrite rules:

$$(p\ b, \mathrm{e}_1); (p\ \neg b, \mathrm{e}_2) \rightsquigarrow (p, (\!|\mathrm{e}_1\ ;\ \mathrm{e}_2|\!)^b) \qquad \text{when } \mathrm{e}_1, \mathrm{e}_2 \text{ are ciphertexts} \tag{5.1}$$

$$(\!|\mathrm{e}\ ;\ \mathrm{e}|\!)^b \rightsquigarrow \mathrm{e} \tag{5.2}$$

$$(\!|\mathrm{e}_1 \cdot \mathrm{e}_2\ ;\ \mathrm{e}_1 \cdot' \mathrm{e}_3|\!)^b \rightsquigarrow \mathrm{e}_1 \cdot (\!|\mathrm{e}_2\ ;\ \mathrm{e}_3|\!)^b \qquad \text{when } \cdot \equiv \cdot' \tag{5.3}$$

$$(\!|\mathrm{e}_1 \cdot \mathrm{e}_2\ ;\ \mathrm{e}_3 \cdot \mathrm{e}_2|\!)^b \rightsquigarrow (\!|\mathrm{e}_1\ ;\ \mathrm{e}_3|\!)^b \cdot \mathrm{e}_2 \qquad \text{when } \cdot \equiv \cdot' \tag{5.4}$$

$$(\!|[\mathrm{e}_1, \ldots, \mathrm{e}_N]\ ;\ [\mathrm{e}'_1, \ldots, \mathrm{e}'_N]|\!)^b \rightsquigarrow [(\!|\mathrm{e}_1\ ;\ \mathrm{e}'_1|\!)^b, \ldots, (\!|\mathrm{e}_N\ ;\ \mathrm{e}'_N|\!)^b] \qquad \text{when only one of } \mathrm{e}_\mathrm{i} \not\equiv \mathrm{e}'_\mathrm{i} \tag{5.5}$$

$$(\!|\mathrm{e}_1\ ;\ \mathrm{e}_2|\!)^b \rightsquigarrow \mathbf{mux}(b, \mathrm{e}_1, \mathrm{e}_2) \qquad \text{when no other rule applies} \tag{5.6}$$

Rule (5.1) identifies adjacent paths that differ only in their final condition, and replaces them with a single path that computes $(\!|\mathrm{e}_1\ ;\ \mathrm{e}_2|\!)^b$, which should be thought of as a "lazy mux" that attempts to pull out as much repeated computation as possible between e_1 and e_2 before branching on b. Rules (5.2)-(5.5) give semantics to the lazy mux operator $(\!|\cdot\ ;\ \cdot|\!)^{\cdot}$. Rules (5.3) and (5.4) allow pulling out a common operand of two arithmetic expressions. Rule (5.5) allows distributing the lazy mux through the elements of two arrays, when all but one of the positions in the arrays contain identical expressions (note that rule (5.2) means that there will only be one mux in the resulting expression). Finally, rule (5.6) describes how to interpret a lazy mux that cannot be simplified any further.

These rewrite rules are mostly standard, but there are a few interesting things to note:

- Rule (5.1) only applies if both e_1 and e_2 are ciphertexts, since otherwise we end up replacing plaintext computation with ciphertext computation.

- We conservatively choose not to include a rule like $(\!|\mathrm{e}_1 \cdot \mathrm{e}_2\ ;\ \mathrm{e}_3 \cdot \mathrm{e}_4|\!)^b \rightsquigarrow (\!|\mathrm{e}_1\ ;\ \mathrm{e}_3|\!)^b \cdot (\!|\mathrm{e}_2\ ;\ \mathrm{e}_4|\!)^b$, since if e_1 and e_3 or e_2 and e_4 don't contain sufficient repeated work, this

just has the effect of producing an extra branch; The condition on rule (5.5) exists for the same reason[8].

- The rules only match on *structural equality* equality, rather than *logical equivalence* (e.g., the expression $(\!|a + b \ ; \ b + a|\!)^b$ cannot be simplified to $a + b$ even though $+$ is commutative). Note that structural equality is usually sufficient, since whenever the unfolding operator $[\![\cdot]\!]_U$ results in repeated work, it creates structurally identical expressions anyway.

Note that these rules are conservative in when they choose to rewrite. This allows us to avoid incorporating heavy-weight techniques like equality saturation and appealing to sophisticated cost models. In the absence of a sophisticated cost model, the rules as written can never introduce additional muxes, and thus prevent degrading performance.

LabelExpr	e	::=	n	numeric literal
		|	**ptxt**(n)	plaintext literal
		|	**ctxt**(n)	ciphertext input
		|	$\mathrm{e}_1 \cdot \mathrm{e}_2$	arithmetic
		|	**mux**(b, e_1, e_2)	multiplexer
BranchingCondition	b	::=	$\mathrm{e}_1 \equiv \mathrm{e}_2$	equality
		|	$\mathrm{e}_1 < \mathrm{e}_2$	inequality
Tree	t	::=	**branch**(b, t_1, t_2)	internal node
		|	**label**(e)	leaf node
Forest	f	::=	t_1 ... t_N	sequence of trees

Figure 5.8. Grammar for decision forests

5.2.6 Generating Code

The final 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.

[8]↑These rewrite rules pull out as much computation as possible under the restriction that they are not allowed to generate additional muxes; It is possible that this leaves some optimization opportunities on the table.

The basic code generation strategy starts by extracting a *decision forest* (Figure 5.8) from a path forest using the following rewrite rules:

$$
\begin{aligned}
[\![(b\ p_1, \mathrm{e}_1); \ldots; (b\ p_N, \mathrm{e}_N); (\neg b\ p'_1, \mathrm{e}'_1); \ldots; (\neg b\ p'_M, \mathrm{e}'_M)]\!]_P &\triangleq \mathbf{branch}(b, [\![(p_1, \mathrm{e}_1); \ldots; (p_N, \mathrm{e}_N)]\!]_P, \\
&\qquad [\![(p'_1, \mathrm{e}'_1); \ldots; (p'_M, \mathrm{e}'_M)]\!]_P) \\
[\![(\emptyset, \mathrm{e})]\!]_P &\triangleq [\![\mathrm{e}]\!]_E \\
[\![\mathbf{mux}(b, \mathrm{e}_1, \mathrm{e}_2)]\!]_E &\triangleq \mathbf{branch}(b, [\![\mathrm{e}_1]\!]_E, [\![\mathrm{e}_2]\!]_E) \\
[\![\mathrm{e}]\!]_E &\triangleq \mathbf{label}(\mathrm{e})
\end{aligned}
$$

These rewrite rules are defined in two operators: $[\![\cdot]\!]_P$ extracts a decision tree from a path forest by recursively identifying paths that start with a common first condition, and $[\![\cdot]\!]_E$ continues the extraction by turning top-level muxes into decision tree branches.

COIL then vectorizes together all the branching conditions of the tree (i.e. every expression that appears as b in **branch**(b, t_1, t_2)), and all the expressions that appear in the labels, and finally uses the COPSE algorithm [49] to generate vectorized code that executes this decision tree to produce the final program result. Section 5.3.1 describes the details of the COPSE algorithm, as well as how we adapt it to fit our needs.

5.3 Implementing COIL

In this section, we discuss the implementation of COIL: the specifics of how we use COPSE (Chapter 4) and our choice of FHE scheme.

5.3.1 Efficient Decision Tree Evaluation via COPSE

Recall from Chapter 4 that 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 (Section 2.3.1). 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 5.9. In particular, we first vectorize together all the `exp1` and (separately) all the `exp2` (e.g. we compute 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.

Additionally, we vectorize together the computations of each label (e.g. we compute the vector `[2a-b; b-2a; 2b-a; a-2b]`), and use the decision result from COPSE to select the correct element of the label vector. For vectorizing the expressions in the conditionals and the labels we use Coyote (Chapter 3).

5.3.2 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 batching* (Section 2.3.1), 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 *integers*[9], 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.

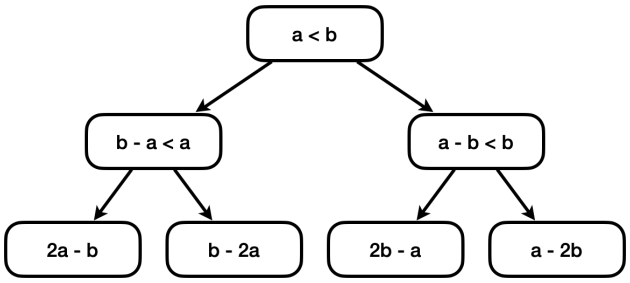


Figure 5.9. Decision tree computing a bounded GCD

5.4 Evaluating COIL

To determine the effectiveness of the compilation techniques presented in this chapter, 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[10]
- **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

[9]↑Technically, CKKS ciphertexts encrypt rational numbers with respect to some variable precision.
[10]↑Our naive implementations are manually transliterated from the COIL surface language to C++. An example of this translation is shown in Figure 5.11b.

- **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.

5.4.1 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

Each benchmark is implemented in COIL's surface language (Figure 5.4), and compiled down to calls to the HElib library [43] using the mod-2 BGV scheme.

The compilation times for each benchmark are reported in Table 5.2, broken down by stage. Notice that with the exception of `merge` and `filter`, the compilation time is negligible (on the order of a few hundred milliseconds). The high compilation times can be attributed

to the combinatorial explosion of paths, a phenomenon which is discussed in more detail in Section 5.4.3, although it is worth noting that in both cases the re-folding mitigates the path explosion reasonably well.

Table 5.1. Speedups of COIL over naive and expert implementations. Note that some numbers are missing: the naive merge times out, and we do not have an expert implementation available for second-price auction.

Benchmark	COIL Time (s)	Naive		Expert	
		Time (s)	Speedup	Time (s)	Speedup
`linear_index`	2.92	67.66	23.16×	5.06	1.73×
`log_index`	1.48	36.34	24.55×	5.06	3.42×
`associative_array`	1.59	18.67	11.75×	5.66	3.56×
`filter`	1.01	17.9	17.76×	0.85	0.85×
`merge`	11.55	—	—	51.32	4.44×
`sp_auction`	15.4	833.93	54.13×	—	—

The time each benchmark takes to run is reported in Figure 5.10, and the speedups are shown in Table 5.1. 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 5.4.3 analyzes where COIL's speedups come from on this benchmark.

Table 5.2. COIL compile times broken down by stage

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

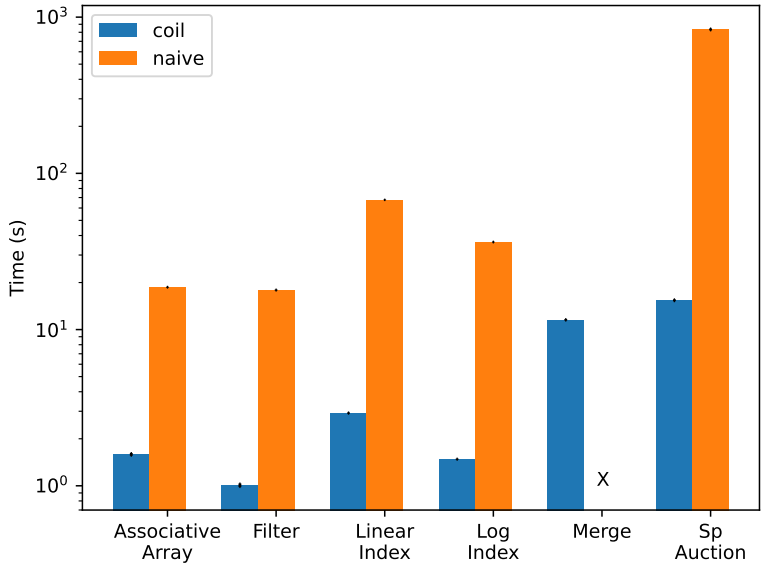


Figure 5.10. Running time of each benchmark. The reported COIL times include both the online and offline phase. Naïve `merge` times out after 30 minutes.

5.4.2 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 5.1.3 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` [55].

Table 5.1 shows a comparison of COIL's execution time to each of these custom protocols. Notice that with the exception of `filter`, the COIL version *outperforms* its associated custom protocol! Even for `filter`, although the path unfolding *does* produce an exponential number of paths, all of these end up refolded together, and the final generated code is nearly identical to the expert implementation.

Furthermore, we see significant speedups on other benchmarks. This is, again, unsurprising for secure array indexing and associative array lookup: Recall from Section 5.1.3 that

```
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++

Figure 5.11. 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.

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 [49]. Explaining the speedup for `merge` is far more interesting, and it is worth taking some time to dissect this benchmark properly.

5.4.3 Where do COIL's speedups come from?

Figure 5.11 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 5.11b 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 5.10).

What does COIL do differently?

The COIL evaluation strategy transforms the snippet in Figure 5.11a 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 [49]. 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.

5.4.4 The advantage of decision tree conversion

Recall that one of the advantages of restructuring a program into a path forest is the ability to then convert the path forest into a decision tree and apply more specialized decision forest inference algorithms such as COPSE [49]. Because it is difficult to disentangle conditionals from other computation in the baseline code, we cannot directly measure the speedup we get from being able to convert the conditionals into a decision tree. However, we can estimate it as follows: The cost of evaluating the conditionals can reasonably be bounded by the cost of sequentially evaluating the decision tree that COIL generates. While we do not have access to an efficient sequential decision tree implementation, we know from COPSE that parallelizing and vectorizing decision tree inference yields a speedup of $\sim 5\times$. As Table 5.3 shows, even after this speedup, for most of our benchmarks the vast majority of the execution time is spent evaluating the decision tree itself[11]. Hence, we see that once COIL unfolds paths, performs pruning and specializing, and refolds them, almost all time is spent in evaluating conditionals. We can therefore estimate the benefit of decision tree conversion plus efficient inference as approximately $5\times$. That leaves the additional $2\times$-$10\times$ speedup over the baseline attributable to the specialization and pruning enabled by path forest conversion

[11]↑The exceptions are `filter` and `sp_auction`. For `filter`, the re-folding pass figures out that the optimal strategy involves just directly muxing, and thus there are no branches in the resulting decision tree; for `sp_auction` only some of the branches get folded into muxes.

Table 5.3. Breakdown of time spent evaluating decision tree branches vs. labels in the generated code

Benchmark	Branches (ms)	Labels (ms)
`linear_index`	2840	5
`log_index`	1710	4
`sp_auction`	6930	9530
`merge`	12400	30
`filter`	0	920
`associative_array`	2070	3

5.5 Scaling & Other Concerns

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, as well as how our techniques compare to classical path-sensitive optimizations.

5.5.1 Path Explosion

The re-folding transformation from Section 5.2.5 helps control the number of paths in the generated code. However, the compiler still has to visit the exponentially many paths produced by the normalization procedure described by Figure 5.6. Thus, the path explosion problem still shows up at *compile time*, even if it does not at execution time. This should be unsurprising: any compiler that aggressively tries to use path-dependent information will have to deal with the path explosion problem.

One might consider a strategy that *entirely avoids* unfolding branches that do not result in a benefit from pruning and specialization, and hence are not "worth it." In fact, this corresponds to a well-known technique called *control flow linearization*, in which explicitly branching control flow is replaced with a branching dataflow operation like a **mux**. [56–60] This is already possible in the COIL language: A programmer who notices that a particular branch is not worth unfolding can replace that branch with a **mux**. From the perspective of the $[\![\cdot]\!]_U$ operator, a **mux** is just another branchless computation, so it does not generate

an extra path in the forest, and hence does not contribute to path explosion. Currently, this transformation must be performed manually, as COIL cannot automatically identify which branches are worth unfolding; we believe that developing this analysis is an interesting future research direction to pursue.

5.5.2 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.

5.5.3 Other Path-Sensitive Analyses

The notion of using path-sensitive information to optimize programs is by no means unique to COIL; indeed, *specialization* can be thought of as a path-sensitive version of constant propagation, *pruning* is similar to identifying infeasible paths and doing dead code elimination, and the normalization rules implemented in $[\![\cdot]\!]_U$ are essentially *partially evaluating* the input program with respect to plaintext values [54, 61–64]. COIL differs from these traditional techniques in two key ways. First, COIL is much more aggressive than traditional path-sensitive dataflow analyses. Specializing with respect to *plaintext values* instead of just *compile-time constants* means more path-dependent information is available, and thus more computation can be specialized, and more paths can be pruned. Second, rather than treating the control-flow structure of the program as a given, COIL explicitly unfolds branches to *expose more paths*. While this is normally not a good idea because of

the compile-time impact, ciphertext computations are so expensive that the benefits of being able to eliminate some make this tradeoff worth it.

5.6 Other Ways of Dealing With Control Flow

5.6.1 Compiling Oblivious Control Flow

The Google Transpiler [19] 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 [65], 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 2.3.1), making it unsuitable for vectorization.

It is worth pointing out that circuit-vectorizing compilers like Coyote [50] *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.

5.6.2 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 [56–60, 66–68]. 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 5.5, applying some control flow reduction techniques to programs before using COIL can help further improve performance.

6. COATL

> The couatl is the sign of ultimate good, disciple of Qotal himself. I, Kachin, priest of Qotal, beseech you to listen...
>
> Ironhelm

So far we have seen one example of compiler cryptosystem co-design: We adapted SLP-style vectorization to FHE with Coyote, developed better cryptosystem abstractions to represent branching control flow with COPSE, and finally designed a language with COIL that knows how to take advantage of both of these to more effectively optimize branchy programs. In this chapter we present COATL, which provides a more self-contained example in the context of a Boolean FHE scheme called CGGI.

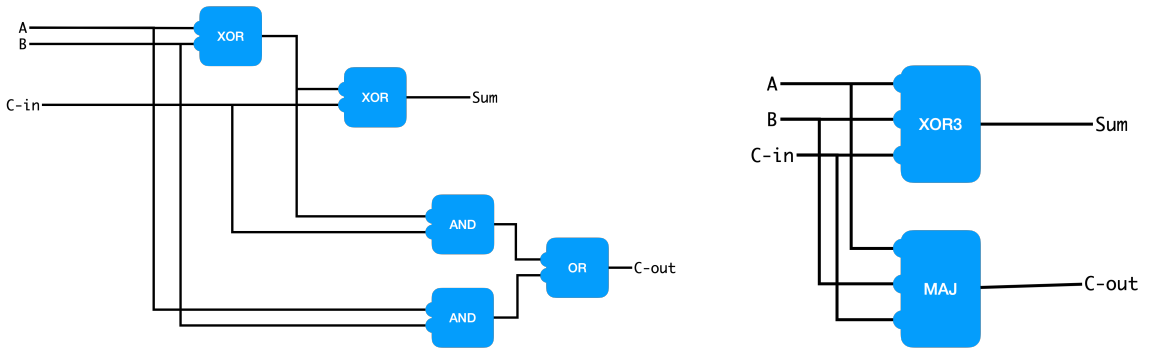
CGGI

Ciphertexts in the CGGI cryptosystem are usually thought of as encryptions of *bits* rather than *integers* (Section 2.3 explores this distinction). Libraries that implement CGGI, such as OpenFHE [37], 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 CGGI actually allows. In particular, CGGI supports an operation called *programmable bootstrapping* (see Section 2.3.3), which allows ciphertexts to be used as indices into arbitrary fixed-size lookup tables; Boolean gates can then be encoded 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 6.1a uses standard Boolean gates (AND, 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 6.1b). (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 6.2.1, most cannot. Existing circuit synthesis techniques therefore cannot use these custom LUTs when generating circuits.

Our key insight in this chapter 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.



(a) An adder using Boolean gates

(b) An adder using custom LUTs

Figure 6.1. Circuits can be made smaller by using custom LUTs instead of traditional Boolean gates

COATL

In this chapter, 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 *already-optimized* Boolean counterparts by up to 1.5×.

6.1 Overview

We begin by providing some intuition behind the key insight of this chapter. We then describe the usual workflow for compiling Boolean FHE programs, and give a high-level overview of how COATL's compilation strategy uses arithmetic LUTs to exploit this insight and "do better".

6.1.1 Some Intuition for Arithmetic LUTs

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.

[1]↑The standard set of encryption parameters in the literature yield eight-row lookup tables [35], though bigger tables are achievable via significantly more expensive parameters. This chapter continues with the convention of eight-row lookup tables, but its ideas are applicable to other parameter sets and table sizes as well.

As mentioned in Section 2.3.3, 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, p-row lookup tables work by computing a ($\mathbb{Z}/p$-)linear combination of the inputs, 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 lookup tables *arithmetic LUTs*, 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.

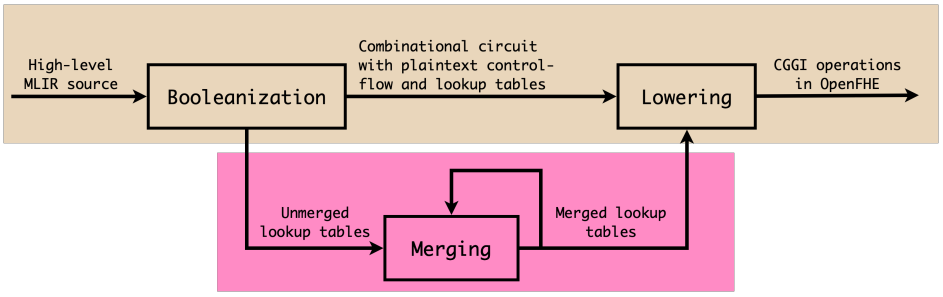


Figure 6.2. Overview of Boolean FHE workflow. The red highlight denotes COATL's workflow

6.1.2 Compiling Boolean FHE Programs

A common strategy for compiling Boolean FHE programs—as illustrated in beige in Figure 6.2, and implemented in, for example, HEIR and the TFHE Transpiler [19, 36]—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 6.4a, which adds two encrypted 8-bit integers, as a running example. Note that for readability, Figure 6.4 (and the other code snippets in this chapter) are presented using a simplified syntax that nevertheless corresponds structurally to the actual frontend MLIR we use (see Section 6.3). 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 [19, 56, 58][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 [22], 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 6.4b 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]`.

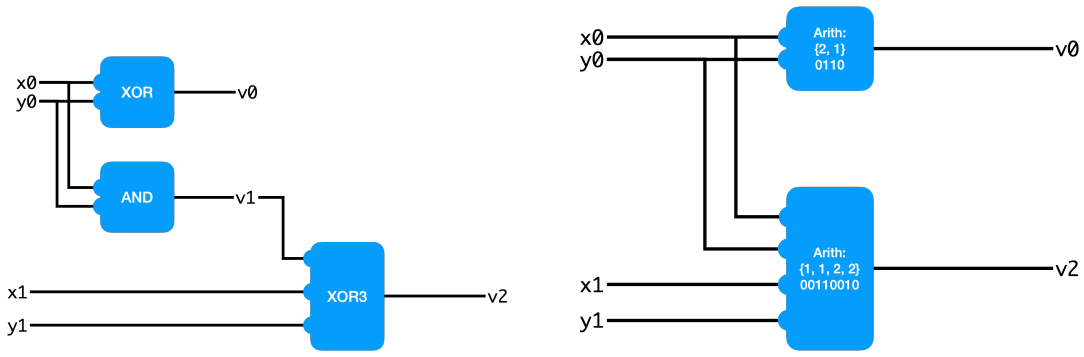
[2]↑By default, we choose not to fully unroll every loop with plaintext bounds, as this can sometimes result in very large circuits. However, the programmer can, to some extent, control whether specific loops get unrolled (see Section 6.3.1).
[3]↑Currently, the set of supported operations includes integer 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.

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 [37]. Similar to Booleanization, this pass preserves all the control flow present in the original circuit, so the generated C++ may contain branches on plaintext values and loops with plaintext bounds. Figure 6.4c 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

6.1.3 Doing Better with COATL



(a) Schematic of Figure 6.4c (b) Schematic of Figure 6.5b

Figure 6.3. COATL merges the `AND` with the `XOR3` to produce a smaller circuit

As depicted by the red highlight in Figure 6.2, 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 6.2.

In the example in Figure 6.4b, 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 6.5a. This process is also shown pictorially in Figure 6.3. 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 6.2 more carefully addresses the general case.

6.2 Building Circuits with Arithmetic LUTs

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?

[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 6.2.1. Similarly, we 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.

```
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

Figure 6.4. Adding two encrypted 8-bit integers

```
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

Figure 6.5. Applying COATL to the booleanized circuit in Figure 6.4b

6.2.1 Arithmetic LUT Formalism

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

Figure 6.6. Truth tables can be modeled using arithmetic LUTs

In this section, we develop the notion of an *arithmetic lookup table* (introduced informally in Section 6.1) 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*:

$$k = \left(\sum_{\mathrm{i}} a_{\mathrm{i}} x_{\mathrm{i}} \right) \bmod R$$

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 6.6a: Any truth table can be modeled as an arithmetic LUT with the same outputs, and the coefficient sequence $\mathbf{a} = (2^{n-1}, 2^{n-2}, \ldots, 2^1, 2^0)$. In fact, recall from Section 2.3.3 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 instead of eight, by setting $\mathbf{a} = (1, 1, 1)$

and $\mathbf{y} = (0, 1, 1, 1)$ as in Figure 6.6b. 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 to CGGI operations.

For a more complex example, consider the truth table in Figure 6.7 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 6.7 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: 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.

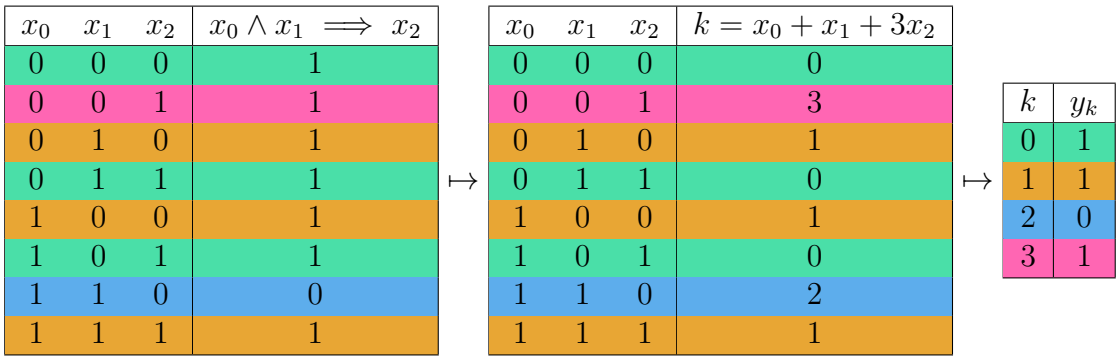
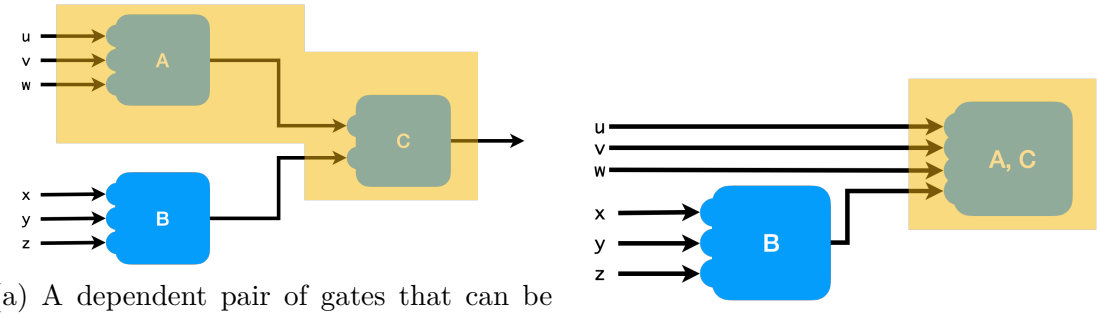


Figure 6.7. A truth table can be mapped into a smaller arithmetic LUT by partitioning its rows, and then finding a linear combination that distinguishes the partitions

6.2.2 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 [21, 22, 36], 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 6.8a). Consider the example circuit in Figure 6.8. 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 6.2.1), then we can represent the same circuit with only two gates instead of three, as shown in Figure 6.8b. 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 6.2.3 discusses how COATL actually identifies pairs to merge.



(a) A dependent pair of gates that can be merged

(b) After merging the dependent pair

Figure 6.8. Merging can yield circuits with fewer gates

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 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 4 shows pseudocode for each step.

Canonicalization

Consider the gates $c = g_1(a, b)$ and $\mathrm{e} = g_2(a, c, d)$. Naïvely merging these gives $\mathrm{e} = g_{1,2}(a, a, b, d)$ 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 $\mathrm{e} = g_{1,2}(a, b, d)$. 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.

Algorithm 4: Merging gates into an arithmetic LUT

Algorithm `BuildLUT(`*g1, g2*`)`

dedupInputs ← `Canonicalize(`*g1, g2*`)`;
ones, zeros ← `Enumerate(`*g1, g2, dedupInputs*`)`;
variables ← $\{a_{\mathrm{i}} \ : \ \mathrm{i} \in \text{inputs}\}$;
constraints ← $\{-R < a_{\mathrm{i}} < R \ : \ a_{\mathrm{i}} \in \text{variables}\}$;
for *o* ← *ones* **do**
  for *z* ← *zeros* **do**
    constraints.add($\sum a_{\mathrm{i}} o_{\mathrm{i}} \neq \sum a_{\mathrm{i}} z_{\mathrm{i}} \pmod R$);
for *o* ← *ones* **do**
  constraints.add($-R < \sum a_{\mathrm{i}} o_{\mathrm{i}} < R$);
for *z* ← *zeros* **do**
  constraints.add($-R < \sum a_{\mathrm{i}} z_{\mathrm{i}} < R$);
if *coefficients* ← `ILPSolve(`*constraints, variables*`)` **then**
  return *coefficients*;
return *error*;

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.

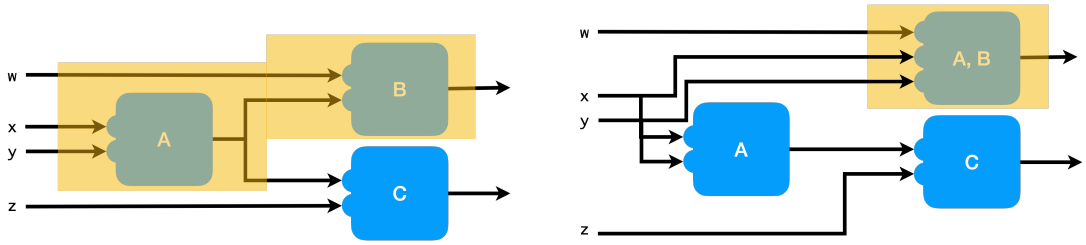
Coefficient Synthesis

We query an ILP solver [69] 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:

- 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 6.5a. To reduce the number of expensive solver calls we make, results are cached using the rows of the uncompressed LUT as a key.



(a) Before (b) After

Figure 6.9. Merging a gate with multiple consumers only shrinks the circuit if all consumers are merged

[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.

6.2.3 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 6.9, 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 2.3.3) 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 6.2.2), we instead use a "complexity 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

[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.

the next iteration. Pseudocode for this procedure is shown in Algorithm 5. Note that the `CanBuildLUT` procedure call implements the steps described in Section 6.2.2 (in particular, the call to the ILP solver), and `PerformMerge` actually updates all the consumers and deletes the now-unused producer.

Algorithm 5: 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 ← ∑_{c∈consumers} Complexity(c.inputs);
    new ← ∑_{c∈consumers} Complexity(c.inputs \ gate.output ∪ gate.inputs);
    return new − old
```

6.2.4 A Small Example

For a concrete example of COATL's LUT merging procedure, consider the circuit schematic in Figure 6.3a. Algorithm 5 starts by identifying the `AND` gate as a candidate for merging, with the `XOR3` gate as its only consumer, at which point we invoke Algorithm 4 to determine whether the pair can be successfully merged into an arithmetic LUT with $R = 8$ rows (recall that the value of R is determined by the encryption parameters and is hence fixed).

Algorithm 4 sets up the ILP problem as follows:

- The deduplicated set of inputs is $\{x_0, y_0, x_1, y_1\}$
- Since there are four inputs, we need four coefficients (a_0, a_1, a_2, a_3)

- We evaluate the merged function: `(x0 AND y0) XOR x1 XOR y1` on all 16 input configurations; **ones** is the set of all configurations on which the function evaluates to `true`, and **zeros** is the set of configurations on which the function evaluates to `false`:
  - **ones** $= \{0001, 0010, 0101, 0110, 1001, 1010, 1100, 1111\}$
  - **zeros** $= \{0000, 0011, 0100, 0111, 1000, 1011, 1101, 1110\}$
- For each pair of bitstrings in **ones** $\times$ **zeros**, we generate an inequality constraint that prevents these two bitstrings from being mapped to the same row. For example, the pair $(0101, 1000)$ generates the constraint $a_1 + a_3 \neq a_0$. This step generates 64 constraints.
- The sum of every subset of coefficients is constrained to be within the range $(-R, R)$. This step generates an additional 16 constraints; for example, $-R < a_0 < R$ and $-R < a_0 + a_1 < R$.

We pass the ILP problem constructed above to a solver, which succeeds and returns the coefficient values $(a_0, a_1, a_2, a_3) = (1, 1, 2, 2)$. Finally, we use these coefficients to build the lookup table:

- Each bitstring maps to a particular row of the LUT as determined by the coefficients. For example, 0110 maps to row $a_1 + a_2 = 3$, and 1011 maps to row $a_0 + a_2 + a_3 = 5$.
- All the bitstrings that map to a given row of the lookup table are guaranteed to evaluate to the same value, which we set as the output corresponding to that row. In our example, rows 2, 3, and 6 map to an output of 1, and the rest map to an output of 0.

Since the merging COATL succeeded, we delete the original two gates and replace them with the generated arithmetic LUT $\mathbf{a} = (1, 1, 2, 2); y = (0, 0, 1, 1, 0, 0, 1, 0)$, as shown in Figure 6.3b.

6.3 Implementation Details

COATL is implemented on top of HEIR[36], 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 6.2. 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.

6.3.1 Unrolling Secret Loops

Recall from Section 6.1.2 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; 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 6.10: by placing the `secret` block *inside* the loop body in Figure 6.10a, 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 6.10b, it is fully unrolled and Booleanized into a single circuit that computes every iteration[8].

6.3.2 OpenFHE Code Generation

COATL targets the OpenFHE [37] 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

[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.

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, 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.

```
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

Figure 6.10. By changing the placement of the `secret` block, the programmer can control whether or not the loop gets unrolled.

6.4 On Scalability

6.4.1 Algorithmic Complexity

ILP Problem Size

Recall from Section 6.2.4 that the size of the ILP problem (i.e. the number of constraints) depends exponentially on the arity of the gate after merging: an arity of n allows for 2^n possible input configurations, which generates $O(4^n)$ total constraints to separate the ones from the zeros. However, the *maximum possible arity* of any gate is limited by the number of rows allowed in a lookup table: indeed, no gate with more than 7 inputs can be successfully compressed into an 8-row LUT. Thus, once a set of encryption parameters (and consequently, the maximum LUT size) has been fixed, the *size* of the ILP problem effectively remains a constant. Furthermore, in practice, the maximum LUT size is usually fairly small (e.g. 4 or 8), so the ILP problem rarely grows beyond tractable limits. Note, in particular, that *the complexity of a single ILP solver call does not depend on the circuit size.*

Overall Complexity

The complexity of COATL's merging procedure is roughly linear in the number of "edges" in the circuit: every edge represents a candidate pair of gates to be merged, and potentially incurs a call to the ILP solver. Furthermore, since COATL uses a "greedy" merging heuristic (and importantly, never backtracks after deciding to merge a pair of gates), a single edge is never visited more than once. Finally, since the total number of edges is bounded above by a constant multiple of the number of gates (in particular, since the input circuit consists of gates with a fixed arity), *COATL's overall complexity is linear in the size of the input circuit.*

6.4.2 Applying COATL to Subcircuits

Although COATL has an asymptotically linear complexity, a single call to an ILP solver can still be relatively expensive. Thus, for a sufficiently large input circuit, exhaustively trying to merge every candidate pair can lead to prohibitively long compilation times.

To get around this limitation, we note that since gates can be merged independently of each other, it is always possible "kernelize" by applying COATL to *smaller subcircuits* in a larger circuit. Large circuits often contain multiple copies of a smaller kernel, so by applying COATL to each kernel once and replacing each copy with its optimized version, we no longer have to visit the entire circuit, and hence can optimize it in sublinear time.

Identifying Subcircuits

A more sophisticated compiler analysis of a given circuit could automatically identify and extract repeated subcircuits to use in the "kernelization" trick described above. While we consider this an interesting direction for future research, COATL does not currently implement this analysis. Instead, it requires the programmer to manually identify a set of common kernels, use COATL to generate a library of efficient implementations of these kernels (as described in Section 6.3.2), and then invoke this library to implement their application. We evaluate the effectiveness of this approach in Section 6.5.4.

6.5 Evaluating COATL

In this section, we evaluate the effectiveness of COATL with 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.

- **RQ4: Is kernelization an effective strategy for optimizing larger circuits?** We implement two kernelized versions of 4×4 matrix multiplication—one using separate "add" and "multiply" kernels, and another using a "multiply-accumulate" kernel—and compare results.

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.

6.5.1 Efficiency of 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) [51, 52, 70–72] at different input sizes.

Table 6.1. Optimized and unoptimized benchmark run time, in milliseconds

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 evaluate COATL on the following set of benchmarks:

- ADD: A 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. An MPC algorithm that determines whether a (secret) key exists in a set. We evaluate retrieval from 8-element sets consisting of 8-bit integers.
- PSI: Private set intersection. An 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.

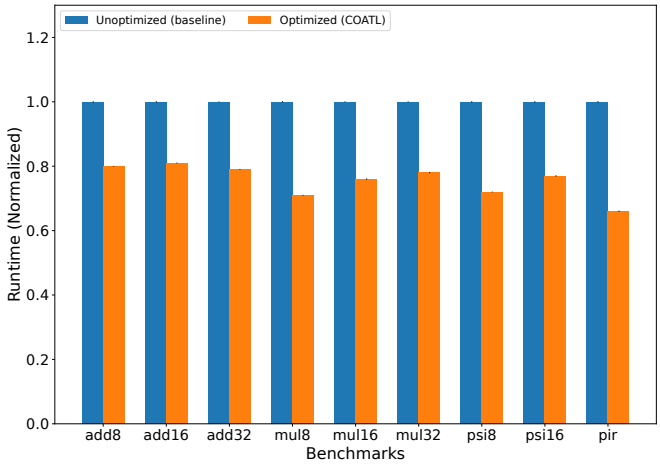


Figure 6.11. Baseline vs COATL run times, normalized to baseline. 95% confidence intervals are plotted on the bar graph, but they are miniscule.

Each benchmark is written in HEIR's `secret` dialect and lowered to OpenFHE C++ code (Section 6.3.2). The C++ code is then compiled and benchmarked. We compare the run times of COATL- and baseline-generated code in Figure 6.11. The run time numbers are plotted normalised to the baseline; the absolute run time numbers and the relative speedup is listed in Table 6.1.

Table 6.2. Optimized and unoptimized benchmark gate counts

Benchmark	Baseline gate count	COATL gate count	Reduction
`add8`	15	11	73%
`add16`	31	24	77%
`add32`	63	50	79%
`mul8`	87	62	71%
`mul16`	401	303	76%
`mul32`	1627	1260	77%
`psi8`	138	98	71%
`psi16`	678	526	78%
`pir`	180	121	67%

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 the use of arithmetic LUTs instead of Boolean truth tables does not result in gates that are more expensive to evaluate: Indeed, as shown in Tables 6.1 and 6.2, the reduction in gates that COATL achieves is strongly correlated with improvements in run time.

6.5.2 Impact on Compilation Time

Here, we compare the compilation times for our benchmarks with and without COATL; the results are shown in Table 6.3. For the baseline, we report the time taken by the entire HEIR pipeline (from high-level code to an optimized circuit), and the time take to compile the generated OpenFHE code. For `HEIR+COATL`, we also report the total amount of time COATL spends in the solver, accumulated across all solver calls. Note that this number is a part of the reported `HEIR+COATL` time, not in addition to it.

Table 6.3. Compile time statistics (in seconds) of unoptimized vs optimized benchmarks. Note that the reported COATL time includes the base HEIR compilation as well as the solver time

Benchmark	Baseline (s)			COATL (s)			
	HEIR	OpenFHE	Total	COATL	Solver	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

We see that most of the overhead of COATL comes from calls to the ILP solver. Furthermore, for larger benchmarks the overall compilation time is dominated by the time it takes to compile the generated OpenFHE code; in fact, this time is *reduced* when using COATL, because the generated circuits are smaller.

6.5.3 Scaling to Larger Inputs

To evaluate how well the optimization scales to different input sizes, we look at the speedup data in Table 6.1. 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 6.1, 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.

Table 6.4. Kernel and Total gate counts of two different kernelizations of 4×4 matrix multiply

Kernel	Kernel gate count		Total gate count		Reduction
	Baseline	COATL	Baseline	COATL	
`ADD+MUL`	15+87	11+62	6528	4672	1.40×
`MAC`	100	73	6400	4672	1.37×

Table 6.5. Compilation and running times of kernelized 4×4 matrix multiply

Kernel	Compilation time (s)		Run time (s)		
	Baseline	COATL	Baseline	COATL	Speedup
`ADD+MUL`	8.5	30.3	151	107.1	1.41×
`MAC`	7.2	29.1	141	103	1.37×

6.5.4 Kernelization

We implement a 4×4 matrix multiplication by kernelizing it two different ways: One implementation uses an 8-bit adder and an 8-bit multiplier as its kernels, and the other uses a single "multiply-accumulate" kernel that computes $f(x, y, z) = xy + z$. Table 6.4 shows the gate counts of these kernels, as well as the total gate counts of the entire matrix multiplication, and Table 6.5 shows the compilation and running times. We find that the speedups COATL achieves on kernelized circuits are comparable to those on our other benchmarks. Furthermore, we note that by kernelizing, COATL is able to optimize much larger programs at a reasonable compile time cost. When we tried compiling a non-kernelized version of the matrix multiplication it took about 70× longer than the kernelized version, and resulted in roughly the same number of gates. Not only does optimizing the circuit take longer, it creates very large code and creates other problems–the generated C++ code caused `gcc` to crash[9] when trying to compile it, so we were unable to collect timing information.

[9]↑This crash is caused by the circuit being huge in general, and not something specific to the code COATL generates.

6.5.5 Impact of Solver Timeout

The HEIR compiler framework ships with a solver [69] which we use to solve the ILP problem described in Section 6.2.2. When profiling COATL's compilation procedure, we find that the compilation time is dominated by the time spent in the solver as shown in Table 6.3. Recall from Section 6.2.3 that the solver is configured with a timeout, and reports an UNSAT result if it fails to find coefficients within that time. For the evaluations in Section 6.5.1, the timeout is set to 500 milliseconds. This timeout is sufficient to avoid missing any optimization opportunities—experimenting with larger timeouts does not produce more-optimized circuits.

7. FUTURE WORK

> The trouble with having an open mind, of course, is that people will insist on coming along and trying to put things in it.
>
> Terry Pratchett, Diggers

This dissertation describes a few problems in the world of privacy-preserving computing, and shows how the philosophy of compiler cryptosystem co-design can be used to improve the state-of-the-art for each of them. Having hopefully justified the space as one sufficiently rich to explore, we conclude by presenting a handful of possible avenues we find interesting for future research.

7.1 Compiler Cryptosystem Choreography

We have, up to this point, implicitly fixed a particular evaluation protocol for FHE computations between a client ("Alice") and an evaluator ("Bob"):

1. Alice and Bob agree on a description of a function to execute (the "circuit")
2. Alice encrypts all her private inputs using some homomorphic scheme, and sends the ciphertexts to Bob, along with her public inputs in plaintext
3. Bob uses the homomorphic scheme to evaluate the circuit over his and Alice's public and private inputs, and sends the (ciphertext) result back to Alice
4. Alice decrypts the final result

In particular, we have made two key assumptions about the computation being performed: First, that Alice's security policy can be fully captured by merely defining which of her inputs should be private; Second, that Bob must compute the entire circuit in "one shot" before sending the results back to Alice (i.e. that the protocol must be "non-interactive").

In this section, we argue that this perspective, while valuable for simplifying discussions and for capturing some common use-cases of FHE, leaves some interesting opportunities on

the table. For instance, consider the following snippet in which Bob receives two encrypted integers from Alice, homomorphically computes the maximum to branch on, then sends the result of evaluating one of two expensive functions (f and g) back to Alice:

```
let x = recv_ctxt(Alice);
let y = recv_ctxt(Alice);
let z = max(x, y);
if (z < 10) {
    // Some expensive computation
    let w = f(z, x);
} else {
     // Some other expensive computation
    let w = g(z, y);
}
send(Alice, w);
```

The usual evaluation strategy described above forces Bob to evaluate both f and g and then use something like a multiplexer to assign w. However, in practice, we often have more nuanced security policies. Alice might, for example, be okay with $z = \max(x, y)$ being made public, despite x and y being private[1]. We can take advantage of this policy by inserting an interactive reveal operation, in which Bob sends a ciphertext computing $max(x, y)$ (together with a proof that the ciphertext does indeed compute $max(x, y)$) to Alice, who decrypts it and sends back the plaintext z:

[1]↑Such complex policies are common when performing computations over sensitive personal data, where some function of the personally identifiable information might be legal to leak; Another example is $z = H(x)$ where H is some cryptographically secure hash function.

```
let x = recv_ctxt(Alice);
let y = recv_ctxt(Alice);
let z = reveal(Bob, max(x, y));
if (z < 10) {
    let w = f(z, x);
} else {
    let w = g(z, y);
}
send(Alice, w);
```

Adding this interactive round obviously does not change the semantics of the program; nor does Bob knowing the plaintext value of z violate the security policy. However, it *does* have an impact on performance, since Bob no longer has to perform the comparison ($z < 10$) homomorphically, and thus can avoid evaluating one of f or g!

This example naturally leads to the following research question: *How do we best take advantage of a more precise security policy by relaxing our protocol to allow for interactivity?* While this appears trivial for the example above, we can readily find programs in which it is not; For instance, consider the program below, together with a security policy in which the result of $x < 10$ is allowed to be leaked:

```
let x = recv_ctxt(Alice);
if (x < 15) {
    let y = f(x);
} else {
    let y = g(x);
}
send(Alice, y);
```

Although the condition $x < 10$ never appears, and thus cannot be explicitly leaked, we can first transform the program as follows by inserting a new branch:

```
let x = recv(Alice);
if (x < 10) {
    let y = f(x);
    send(Alice, y);
} else {
    // The original program...
}
```

and then reveal $x < 10$ via an an interactive round. The new program is still correct because $x < 10 \implies x < 15$, but by revealing the result of the first branch Bob can, in some cases, avoid computing g altogether. Indeed, in general, fully exploiting the security policy may require significantly changing the control flow- and dataflow-structure of the original program.

In fact, we can imagine taking this a few steps further still: Rather than restricting Alice to merely decrypting ciphertexts and sending them back to Bob, we can ask her to *perform arbitrary local computations*, over both her own inputs *and* explicitly revealed intermediates, before sending the results back to Bob. In the limit, we could imagine a compilation model in which we generate, not only efficient circuits and cryptosystems, but *entire bespoke choreographies*, precisely tailored to each application and security policy!

7.2 Bootstrapping-Aware Compilation

Our work so far has primarily dealt with the *computational content* of a program; i.e., our circuits consist only of operations that correspond to some computation on plaintexts, like adding/multiplying ciphertexts, evaluating a lookup table, or rotating a vector. In this section, we expand our focus to also consider *bootstrapping*.

Recall from Section 2.3.2 that bootstrapping is a *ciphertext maintenance* operation: It is effectively a "no-op" on the underlying encrypted value, but it performs some useful side-effect, in this case, resetting the ciphertext *level*. Bootstrapping, while generally the slowest operation in any FHE scheme, is crucial for larger applications that require more levels than

are available with a reasonable set of encryption parameters; Thus, the problem of how to optimize bootstraps is the subject of much modern FHE research.

Compiler techniques that deal with bootstrapping typically start with an optimized circuit and then solve a graph problem to determine the optimal places to insert bootstraps. These techniques often have to make some simplifying assumptions to model the semantics of bootstrapping, but perform reasonably well under these assumptions, and crucially, prevent the programmer from having to explicitly reason about bootstraps.

The cryptography community takes a different approach entirely, instead developing novel kinds of "functional" bootstrapping that, instead of being a no-op on plaintext values, are actually capable of performing some limited computation at the same time (in fact, the programmable bootstrapping used to implement the lookup tables in Chapter 6 is exactly an instance of this!) As was already true in Chapter 6, and is even more true for the more limited kinds of functional bootstrapping supported in othe FHE schemes like BFV, using these bootstrapping operations efficiently is a difficult task for non-experts, so we tend to see them in bespoke protocols rather than as targets of compilation.

Applying the philosphy of compiler cryptosystem co-design, we arrive at a natural research question: *What does a compiler look like that reasons about bootstrapping at* every step of the pipeline*, rather than at the end?* In other words, instead of placing bootstraps into an already-optimized circuit, can we make different optimization decisions based on what might yield better bootstrap placements? Can we "restructure" computations in a way similar to COATL, so that natural functional bootstrapping points line up with compatible operations in the circuit?

REFERENCES

[1] D. W. Archer *et al.*, "Ramparts: A programmer-friendly system for building homomorphic encryption applications," in *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, ser. WAHC'19, London, United Kingdom: Association for Computing Machinery, 2019, pp. 57–68, ISBN: 9781450368292. DOI: 10.1145/3338469.3358945. [Online]. Available: https://doi.org/10.1145/3338469.3358945.

[2] A. Rastogi, M. A. Hammer, and M. W. Hicks, "Wysteria: A programming language for generic, mixed-mode multiparty computations," *2014 IEEE Symposium on Security and Privacy*, pp. 655–670, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:5154250.

[3] A. Rastogi, N. Swamy, and M. Hicks, "Wys*: A dsl for verified secure multi-party computations," in *Principles of Security and Trust*, F. Nielson and D. Sands, Eds., Cham: Springer International Publishing, 2019, pp. 99–122, ISBN: 978-3-030-17138-4.

[4] M. Keller, "Mp-spdz: A versatile framework for multi-party computation," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1575–1590, ISBN: 9781450370899. DOI: 10.1145/3372297.3417872. [Online]. Available: https://doi.org/10.1145/3372297.3417872.

[5] Y. Zhang, A. Steele, and M. Blanton, "Picco: A general-purpose compiler for private distributed computation," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, Berlin, Germany: Association for Computing Machinery, 2013, pp. 813–826, ISBN: 9781450324779. DOI: 10.1145/2508859.2516752. [Online]. Available: https://doi.org/10.1145/2508859.2516752.

[6] S. Zahur and D. Evans, *Obliv-c: A language for extensible data-oblivious computation*, Cryptology ePrint Archive, Paper 2015/1153, 2015. [Online]. Available: https://eprint.iacr.org/2015/1153.

[7] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 359–376. DOI: 10.1109/SP.2015.29.

[8] S. Carpov, P. Dubrulle, and R. Sirdey, "Armadillo: A compilation chain for privacy preserving applications," in *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, ser. SCC '15, Singapore, Republic of Singapore: Association for Computing Machinery, 2015, pp. 13–19, ISBN: 9781450334471. DOI: 10.1145/2732516.2732520. [Online]. Available: https://doi.org/10.1145/2732516.2732520.

[9] E. Crockett, C. Peikert, and C. Sharp, "Alchemy: A language and compiler for homomorphic encryption made easy," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 1020–1037, ISBN: 9781450356930. DOI: 10.1145/3243734.3243828. [Online]. Available: https://doi.org/10.1145/3243734.3243828.

[10] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 546–561, ISBN: 9781450376136. DOI: 10.1145/3385412.3386023. [Online]. Available: https://doi.org/10.1145/3385412.3386023.

[11] C. Mendis and S. Amarasinghe, "Goslp: Globally optimized superword level parallelism framework," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 2018. DOI: 10.1145/3276480. [Online]. Available: https://doi.org/10.1145/3276480.

[12] Y. Chen, C. Mendis, M. Carbin, and S. Amarasinghe, "Vegen: A vectorizer generator for SIMD and beyond," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021, Virtual, USA: Association for Computing Machinery, 2021, pp. 902–914, ISBN: 9781450383172. DOI: 10.1145/3445814.3446692. [Online]. Available: https://doi.org/10.1145/3445814.3446692.

[13] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ser. PLDI '00, Vancouver, British Columbia, Canada: Association for Computing Machinery, 2000, pp. 145–156, ISBN: 1581131992. DOI: 10.1145/349299.349320. [Online]. Available: https://doi.org/10.1145/349299.349320.

[14] Y. Bao *et al.*, "Haccle: Metaprogramming for secure multi-party computation," in *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2021, Chicago, IL, USA: Association for Computing Machinery, 2021, pp. 130–143, ISBN: 9781450391122. DOI: 10.1145/3486609.3487205. [Online]. Available: https://doi.org/10.1145/3486609.3487205.

[15] C. Acay, R. Recto, J. Gancher, A. C. Myers, and E. Shi, "Viaduct: An extensible, optimizing compiler for secure distributed programs," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021, Virtual, Canada: Association for Computing Machinery, 2021, pp. 740–755, ISBN: 9781450383912. DOI: 10.1145/3453483.3454074. [Online]. Available: https://doi.org/10.1145/3453483.3454074.

[16] C. Boura, N. Gama, M. Georgieva, and D. Jetchev, *Journal of Mathematical Cryptology*, vol. 14, no. 1, pp. 316–338, 2020. DOI: doi:10.1515/jmc-2019-0026. [Online]. Available: https://doi.org/10.1515/jmc-2019-0026.

[17] W.-j. Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu, "Pegasus: Bridging polynomial and non-polynomial evaluations in homomorphic encryption," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1057–1073. DOI: 10.1109/SP40001.2021.00043.

[18] R. Dathathri *et al.*, "Chet: An optimizing compiler for fully-homomorphic neural-network inferencing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 142–156, ISBN: 9781450367127. DOI: 10.1145/3314221.3314628. [Online]. Available: https://doi.org/10.1145/3314221.3314628.

[19] S. Gorantala *et al.*, *A general purpose transpiler for fully homomorphic encryption*, Cryptology ePrint Archive, Paper 2021/811, https://eprint.iacr.org/2021/811, 2021. [Online]. Available: https://eprint.iacr.org/2021/811.

[20] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, *HECO: Fully homomorphic encryption compiler*, 2023. arXiv: 2202.01649 [cs.CR].

[21] X. Contributors, *XLS: Accelerated HW Synthesis*, https://github.com/google/xls, 2024.

[22] C. Wolf, *Yosys open synthesis suite*, https://yosyshq.net/yosys/.

[23] C. Lattner and J. Pienaar, *Mlir primer: A compiler infrastructure for the end of moore's law*, 2019.

[24] A. Krastev, N. Samardzic, S. Langowski, S. Devadas, and D. Sanchez, "A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption," *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024. DOI: 10.1145/3656382. [Online]. Available: https://doi.org/10.1145/3656382.

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

[26] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, "Porcupine: A synthesizing compiler for vectorized homomorphic encryption," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021, Virtual, Canada: Association for Computing Machinery, 2021, pp. 375–389, ISBN: 9781450383912. DOI: 10.1145/3453483.3454050. [Online]. Available: https://doi.org/10.1145/3453483.3454050.

[27] V. Ding, C. Acay, and A. C. Myers, "An array intermediate language for mixed cryptography," in *FCS*, 2023.

[28] G. Asharov, A. Jain, and D. Wichs, *Multiparty computation with low communication, computation and interaction via threshold FHE*, Cryptology ePrint Archive, Report 2011/613, 2011.

[29] L. Chen, Z. Zhang, and X. Wang, "Batched multi-hop multi-key FHE from ring-LWE with compact ciphertext extension," in *TCC*, 2017.

[30] N. Li, T. Zhou, X. Yang, Y. Han, W. Liu, and G. Tu, "Efficient multi-key FHE with short extended ciphertexts and directed decryption protocol," *IEEE Access*, vol. 7, pp. 56 724–56 732, 2019.

[31] N. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *IACR Cryptology ePrint Archive*, vol. 2011, p. 133, Jan. 2011. DOI: 10.1007/s10623-012-9720-4.

[32] Z. Brakerski, C. Gentry, and S. Halevi, *Packed ciphertexts in lwe-based homomorphic encryption*, 2012.

[33] C. Gentry, "A fully homomorphic encryption scheme," AAI3382729, Ph.D. dissertation, Stanford, CA, USA, 2009, ISBN: 9781109444506.

[34] C. Gentry, S. Halevi, and N. P. Smart, "Better bootstrapping in fully homomorphic encryption," in *Public Key Cryptography – PKC 2012*, M. Fischlin, J. Buchmann, and M. Manulis, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–16, ISBN: 978-3-642-30057-8.

[35] D. Micciancio and Y. Polyakov, *Bootstrapping in FHEW-like cryptosystems*, Cryptology ePrint Archive, Paper 2020/086, 2020. [Online]. Available: https://eprint.iacr.org/2020/086.

[36] H. Contributors, *HEIR: Homomorphic Encryption Intermediate Representation*, https://github.com/google/heir, 2023.

[37] A. A. Badawi *et al.*, *OpenFHE: Open-source fully homomorphic encryption library*, Cryptology ePrint Archive, Paper 2022/915, https://eprint.iacr.org/2022/915, 2022. [Online]. Available: https://eprint.iacr.org/2022/915.

[38] D. Lee, W. Lee, H. Oh, and K. Yi, "Optimizing homomorphic evaluation circuits by program synthesis and term rewriting," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, London, UK: Association for Computing Machinery, 2020, pp. 503–518, ISBN: 9781450376136. DOI: 10.1145/3385412.3385996. [Online]. Available: https://doi.org/10.1145/3385412.3385996.

[39] S. Halevi and V. Shoup, *Algorithms in helib*, Cryptology ePrint Archive, Report 2014/106, https://ia.cr/2014/106, 2014.

[40] P. M. Phothilimthana *et al.*, "Swizzle inventor: Data movement synthesis for gpu kernels," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 65–78, ISBN: 9781450362405. DOI: 10.1145/3297858.3304059. [Online]. Available: https://doi.org/10.1145/3297858.3304059.

[41] *Microsoft SEAL (release 3.7)*, https://github.com/Microsoft/SEAL, Microsoft Research, Redmond, WA., Sep. 2021.

[42] A. Aloufi, P. Hu, H. W. H. Wong, and S. S. M. Chow, *Blindfolded evaluation of random forests with multi-key homomorphic encryption*, Cryptology ePrint Archive, Paper 2019/819, https://eprint.iacr.org/2019/819, 2019. [Online]. Available: https://eprint.iacr.org/2019/819.

[43] S. Halevi and V. Shoup, "Design and implementation of a homomorphic-encryption library," 2012.

[44] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Comput. Theory*, vol. 6, no. 3, Jul. 2014, ISSN: 1942-3454. DOI: 10.1145/2633600. [Online]. Available: https://doi.org/10.1145/2633600.

[45] MLData, *Census income*, https://www.mldata.io/dataset-details/census_income/, 1996 (accessed 2020).

[46] MLData, *Soccer international history*, https://www.mldata.io/dataset-details/soccer_international_history/, 2017 (accessed 2020).

[47] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[48] J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007, ISBN: 978-0-596-51480-8. [Online]. Available: http://www.oreilly.com/catalog/9780596514808/index.html.

[49] R. Malik, V. Singhal, B. Gottfried, and M. Kulkarni, "Vectorized secure evaluation of decision forests," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021, Virtual, Canada: Association for Computing Machinery, 2021, pp. 1049–1063, ISBN: 9781450383912. DOI: 10.1145/3453483.3454094. [Online]. Available: https://doi.org/10.1145/3453483.3454094.

[50] R. Malik, K. Sheth, and M. Kulkarni, "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*, ser. ASPLOS 2023, Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 118–133, ISBN: 9781450399180. DOI: 10.1145/3582016.3582057. [Online]. Available: https://doi.org/10.1145/3582016.3582057.

[51] M. Blanton, A. Kang, and C. Yuan, *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, 2019. [Online]. Available: https://eprint.iacr.org/2019/718.

[52] F. Bayatbabolghani, M. Blanton, M. Aliasgari, and M. T. Goodrich, "Secure fingerprint alignment and matching protocols," *CoRR*, vol. abs/1702.03379, 2017. arXiv: 1702.03379. [Online]. Available: http://arxiv.org/abs/1702.03379.

[53] U. Jørring and W. L. Scherlis, "Compilers and staging transformations," in *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '86, St. Petersburg Beach, Florida: Association for Computing Machinery, 1986, pp. 86–96, ISBN: 9781450373470. DOI: 10.1145/512644.512652. [Online]. Available: https://doi.org/10.1145/512644.512652.

[54] C. A. Herrmann and T. Langhammer, "Combining partial evaluation and staged interpretation in the implementation of domain-specific languages," *Science of Computer Programming*, vol. 62, no. 1, pp. 47–65, 2006, Special Issue on the First MetaOCaml Workshop 2004, ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2006.02.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642306000736.

[55] S. Hong, S. Kim, J. Choi, Y. Lee, and J. H. Cheon, *Efficient sorting of homomorphic encrypted data with k-way sorting network*, Cryptology ePrint Archive, Paper 2021/551, https://eprint.iacr.org/2021/551, 2021. DOI: 10.1109/TIFS.2021.3106167. [Online]. Available: https://eprint.iacr.org/2021/551.

[56] S. Moll and S. Hack, "Partial control-flow linearization," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018, Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 543–556, ISBN: 9781450356985. DOI: 10.1145/3192366.3192413. [Online]. Available: https://doi.org/10.1145/3192366.3192413.

[57] J. Ferrante and M. Mace, "On linearizing parallel code," in *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '85, New Orleans, Louisiana, USA: Association for Computing Machinery, 1985, pp. 179–190, ISBN: 0897911474. DOI: 10.1145/318593.318636. [Online]. Available: https://doi.org/10.1145/318593.318636.

[58] J. Anantpur and G. R., "Taming control divergence in gpus through control flow linearization," in *Compiler Construction*, A. Cohen, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 133–153, ISBN: 978-3-642-54807-9.

[59] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '83, Austin, Texas: Association for Computing Machinery, 1983, pp. 177–189, ISBN: 0897910907. DOI: 10.1145/567067.567085. [Online]. Available: https://doi.org/10.1145/567067.567085.

[60] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanović, "Convergence and scalarization for data-parallel architectures," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013, pp. 1–11. DOI: 10.1109/CGO.2013.6494995.

[61] S. Ding and H. B. K. Tan, "Detection of infeasible paths: Approaches and challenges," in *Evaluation of Novel Approaches to Software Engineering*, L. A. Maciaszek and J. Filipe, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 64–78, ISBN: 978-3-642-45422-6.

[62] K. Winter, C. Zhang, I. J. Hayes, N. Keynes, C. Cifuentes, and L. Li, "Path-sensitive data flow analysis simplified," in *Formal Methods and Software Engineering*, L. Groves and J. Sun, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 415–430, ISBN: 978-3-642-41202-8.

[63] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Commun. ACM*, vol. 19, no. 3, p. 137, Mar. 1976, ISSN: 0001-0782. DOI: 10.1145/360018.360025. [Online]. Available: https://doi.org/10.1145/360018.360025.

[64] N. D. Jones, "An introduction to partial evaluation," *ACM Comput. Surv.*, vol. 28, no. 3, pp. 480–503, Sep. 1996, ISSN: 0360-0300. DOI: 10.1145/243439.243447. [Online]. Available: https://doi.org/10.1145/243439.243447.

[65] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, *TFHE: Fast fully homomorphic encryption over the torus*, Cryptology ePrint Archive, Paper 2018/421, 2018. [Online]. Available: https://eprint.iacr.org/2018/421.

[66] S. Moll and S. Hack, "Partial control-flow linearization," *SIGPLAN Not.*, vol. 53, no. 4, pp. 543–556, Jun. 2018, ISSN: 0362-1340. DOI: 10.1145/3296979.3192413. [Online]. Available: https://doi.org/10.1145/3296979.3192413.

[67] C. Saumya, K. Sundararajah, and M. Kulkarni, "Darm: Control-flow melding for simt thread divergence reduction," in *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '22, Virtual Event, Republic of Korea: IEEE Press, 2022, pp. 28–40, ISBN: 9781665405843. DOI: 10.1109/CGO53902.2022.9741285. [Online]. Available: https://doi.org/10.1109/CGO53902.2022.9741285.

[68] R. Taylor and X. Li, "Software-based branch predication for amd gpus," *SIGARCH Comput. Archit. News*, vol. 38, no. 4, pp. 66–72, Jan. 2011, ISSN: 0163-5964. DOI: 10.1145/1926367.1926379. [Online]. Available: https://doi.org/10.1145/1926367.1926379.

[69] L. Perron, F. Didier, and S. Gay, "The cp-sat-lp solver," in *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, R. H. C. Yap, Ed., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 280, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 3:1–3:2, ISBN: 978-3-95977-300-3. DOI: 10.4230/LIPIcs.CP.2023.3. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2023/19040.

[70] H. Chen, K. Laine, and P. Rindal, *Fast private set intersection from homomorphic encryption*, Cryptology ePrint Archive, Paper 2017/299, 2017. [Online]. Available: https://eprint.iacr.org/2017/299.

[71] H. Chen, Z. Huang, K. Laine, and P. Rindal, *Labeled PSI from fully homomorphic encryption with malicious security*, Cryptology ePrint Archive, Paper 2018/787, 2018. DOI: 10.1145/3243734.3243836. [Online]. Available: https://eprint.iacr.org/2018/787.

[72] F. Boemer, K. Tarbe, and R. Rishi, *Announcing swift homomorphic encryption*, 2024. [Online]. Available: https://www.swift.org/blog/announcing-swift-homomorphic-encryption/.