Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption

Meghan Cowan
Facebook Reality Labs Research
Redmond, WA, USA
meghancowan@fb.com

Deeksha Dangwal
Facebook Reality Labs Research
Redmond, WA, USA
ddangwal@fb.com

Armin Alaghi
Facebook Reality Labs Research
Redmond, WA, USA
alaghi@fb.com

Caroline Trippel
Stanford University
Stanford, CA, USA
trippel@stanford.edu

Vincent T. Lee
Facebook Reality Labs Research
Redmond, WA, USA
vtlee@fb.com

Brandon Reagen
New York University
New York, NY, USA
bjr5@nyu.edu

Abstract

Homomorphic encryption (HE) is a privacy-preserving technique that enables computation directly on encrypted data. Despite its promise, HE has seen limited use due to performance overheads and compilation challenges. Recent work has made significant advances to address the performance overheads but automatic compilation of efficient HE kernels remains relatively unexplored.

This paper presents Porcupine, an optimizing compiler that generates vectorized HE code using program synthesis. HE poses three major compilation challenges: it only supports a limited set of SIMD-like operators, it uses long-vector operands, and decryption can fail if ciphertext noise growth is not managed properly. Porcupine captures the underlying HE operator behavior so that it can automatically reason about the complex trade-offs imposed by these challenges to generate optimized, verified HE kernels. To improve synthesis time, we propose a series of optimizations including a sketch design tailored to HE to narrow the program search space. We evaluate Porcupine using a set of kernels and show speedups of up to 52% (25% geometric mean) compared to heuristic-driven hand-optimized kernels. Analysis of Porcupine's synthesized code reveals that optimal solutions are not always intuitive, underscoring the utility of automated reasoning in this domain.

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454050

CCS Concepts: • **Security and privacy** → *Software security engineering*; • **Software and its engineering** → **Compilers**; *Automatic programming*.

Keywords: homomorphic encryption, vectorization, program synthesis

ACM Reference Format:
Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. 2021. Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454050

1 Introduction

Homomorphic encryption (HE) is a rapidly maturing privacy-preserving technology that enables computation directly on encrypted data. HE enables secure remote computation, as cloud service providers can compute on data without viewing the data's contents. Despite its appeal, two key challenges prevent widespread HE adoption: performance and programmability. Today, most systems-oriented HE research has focused on overcoming the prohibitive performance overheads with high-performance software libraries [36, 44] and custom hardware [40, 42]. The performance results are encouraging with some suggesting that HE can approach real-time latency for certain applications with sufficiently large hardware resources [40]. Realizing the full potential of HE requires an analogous compiler effort to alleviate the code generation and programming challenges, which remain less explored.

Modern ring-based HE schemes pose three programming challenges: (i) they only provide a limited set of instructions (add, multiply, and rotate); (ii) the ciphertext operands are long vectors, on the order of thousands; (iii) ciphertexts have noise that grows as operations are performed and causes decryption to fail if too much accumulates. For instance, the Brakerski/Fan-Vercauteren (BFV) crpytosystem [18] operates on vectors that pack multiple data elements into a

single ciphertext to improve performance. Instructions operating on packed-vector ciphertexts can be abstracted as a SIMD (single instruction, multiple data) instruction set, which introduces vectorization challenges.

To target the instruction set, the programmer must break down an input kernel into SIMD addition, multiply, and rotation instructions, while minimizing noise accumulation. These challenges introduce a complex design space when implementing HE kernels. As a result, HE kernels are currently written by a limited set of experts fluent in "HE-assembly" and the details of ciphertext noise. Even for experts, this process is laborious. As a result, hand-writing HE programs does not scale beyond a few kernels. *Thus, automated compiler support for HE is needed for it to emerge as a viable solution for privacy-preserving computation.*

A nascent body of prior work exists and has investigated specific aspects of compiling HE code. For example, prior work has shown HE parameter tuning, which determines the noise budget, can be automated and optimized to improve performance [3, 12, 15, 16]. Others have proposed mechanisms to optimize data layouts for neural networks [16]. Prior solutions have also used a mix of symbolic execution [3] and rewrite rules [7, 12, 15] for code generation and optimizations for logic minimization (e.g., Boolean logic minimization [12, 28]. Each of these lines of work have advanced the field and addressed notable HE compilation challenges. In contrast to related work (see Section 8), we are the first to automate compiling and optimizing vectorized HE kernels.

In this paper, we propose Porcupine, a synthesizing compiler for HE. Users provide a reference implementation of their plaintext kernel, and Porcupine synthesizes a vectorized HE kernel that performs the same computation. Internally, Porcupine models instruction noise, latency, behavior, and HE program semantics with Quill, a novel HE DSL. Quill enables Porcupine to reason about and search for HE kernels that are (verifiably) correct and minimizes the kernel's cost, i.e., latency and noise accumulation. With Porcupine and Quill, we develop a synthesis procedure that automates and optimizes the mapping and scheduling of plaintext kernels to HE instructions.

Porcupine uses syntax-guided synthesis [2], and operates by completing a sketch, or HE kernel template. We introduce a novel *local rotate* sketch that treats ciphertext rotation as an input to HE add and multiply instructions rather than an independent rotation instruction; this makes the synthesis search more tractable by limiting the space of possible programs. Furthermore, we develop several HE-specific optimizations including rotation restrictions for tree reductions and stencil computations, multi-step synthesis, and constraint optimizations to further improve synthesis run time (details in Section 6).

We evaluate Porcupine using a variety of image processing and linear algebra kernels. Baseline programs are hand-written and attempt to minimize multiplicative and logical depth, the current best practice for optimizing HE programs [3, 12, 28]. For small kernels, Porcupine is able to find the same optimized implementations as the hand-written baseline. On larger, more complex kernels, we show Porcupine's programs are up to 52% faster. Upon further analysis, we find that Porcupine can discover optimizations such as factorization and even application-specific optimizations involving separable filters. Our results demonstrate the efficacy and generality of our synthesis-based compilation approach and further motivates the benefits of automated reasoning in HE for both performance and productivity.

This paper makes the following contributions:

1. We present Porcupine, a program synthesis-based compiler that automatically generates vectorized HE programs, and Quill, a DSL for HE. Porcupine includes a set of optimizations needed to effectively adopt program synthesis to target HE.
2. We evaluate Porcupine using nine kernels to demonstrate it can successfully translate plaintext specifications to correct HE-equivalent implementations. Porcupine achieves speedups of up to 52% (25% geometric mean) over hand-written baselines implemented with best-known practices. We note situations where optimal solutions cannot be found with existing techniques (i.e., logic depth minimization), further motivating automated reasoning-based solutions.
3. We develop a set of optimizations to improve Porcupine's synthesis time and compile larger programs. First, we develop a domain-specific local rotate sketch that considers rotations as inputs to arithmetic instructions, narrowing the solutions space without compromising quality. We further restrict HE rotation patterns and propose a multi-step synthesis process.

2 Homomorphic Encryption Background

This section provides a brief background on homomorphic encryption. We refer the reader to [8–10, 18, 21] for the more technical details of how HE works.

2.1 Homomorphic Encryption Basics

Homomorphic encryption enables arbitrary computation over encrypted data or ciphertexts [20]. This enables secure computation offload where an untrusted third party, such as a cloud provider, performs computation over a client's private data without gaining access to it.

Figure 1 shows a canonical HE system for secure cloud compute. First, the client locally encrypts their data asset x using a private key k. The resulting ciphertext x' is then sent to the cloud where an HE function g' is applied to it. The output of the computation $g'(x')$ is then sent back to

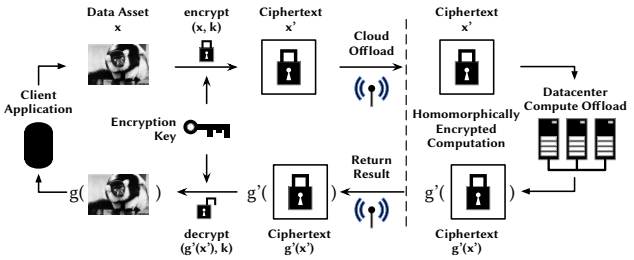

Figure 1. HE system for to an untrusted third party cloud. A plaintext data asset x is encrypted with a key k to generate ciphertext x' and transmitted to the cloud. The cloud service applies HE computation g' to the ciphertext *without* decrypting the data. The result $g'(x')$ is transmitted back to client where decryption yields the result $g(x)$. Lemur image ©Skip Brown, Smithsonian's National Zoo.

the client and decrypted using the same key k to reveal the plaintext output: $g(x)$. HE allows us to define a function g' that operates over ciphertext $x' = \text{encrypt}(x, k)$ such that:

$$\text{decrypt}(g'(x'), k) = g(x)$$

The private key k never leaves the client, ensuring the client's data asset is secure throughout the computation. Additionally, the client does not learn g, which could be a secret that the cloud wants to protect. Porcupine's goal is to synthesize g' given a definition of the kernel g.

This paper focuses on the BFV cryptosystem, a specific HE scheme that targets integers [18]. In the remainder of this section, we provide an overview of the BFV scheme and focus on the vector programming model, instructions, and noise considerations it exposes. For a more technical description see [1, 18].

2.2 BFV

BFV is an HE scheme that operates over encrypted integers. In BFV, integers are encrypted into a ciphertext polynomial of degree N with integer coefficients that are modulo q. A key property of BFV is batching; this allows a vector of up to N integers to be encrypted in a single ciphertext with operations behaving in a SIMD manner.

For the most part, ciphertext polynomials behave as a vector of N slots with bitwidth q. N and q are BFV HE parameters set to provide a desired security level and computational depth, not the number of raw integers that are encrypted. Regardless of whether we encrypt a single integer or N integers in a ciphertext, a vector of N slots is allocated for security purposes. N is required to be a large power of two and is often in the tens of thousands, which makes batching crucial to efficiently utilizing ciphertext space.

Instructions. BFV provides three core ciphertext instructions that behave like element-wise SIMD instructions: SIMD add, SIMD multiply, and SIMD (slot) rotate. Additionally, BFV supports variants of add and multiply that operate on a ciphertext and plaintext instead of two ciphertexts.

Consider two vectors of integers $X = \{x_0, x_1, ..., x_{n-1}\}$ and $Y = \{y_0, y_1, ..., y_{n-1}\}$ with ciphertext representation X' and Y' respectively. SIMD add and multiply both perform element-wise operations over slots. SIMD add computes $add(X', Y')$ such that $\text{decrypt}(add(X', Y')', k) = \{x_0+y_0, x_1+y_1, ..., x_{n-1}+y_{n-1}\}$, where k is the key used for encryption of X' and Y'. Similarly, the SIMD multiply instruction processes $mul(X, Y)$ so that $\text{decrypt}(g'(X', Y'), k) = \{x_0 \times y_0, x_1 \times y_1, ..., x_{n-1} \times y_{n-1}\}$. Note that the underlying operations that implement $add(X', Y')$ and $mul(X', Y')$ over the ciphertext representations are *not* simple vector addition or multiplication instructions.

Rotate. Additionally, HE provides rotate instructions that circularly shift slots in a ciphertext by an integer amount (similar to bitwise rotations). Rotations occur in unison: given a rotation amount, all slots shift by the same amount in the same direction and the relative ordering of slots is preserved. For example, rotating a ciphertext $X' = \{x_0, x_1, x_2, ..., x_{n-1}\}$ by one element to the left returns $\{x_1, x_2, ..., x_{n-1}, x_0\}$.

Note the ciphertext is not a true vector, so slots cannot be directly indexed or re-ordered. Slot rotation is necessary to align slot values between vectors because add and multiply instructions are element-wise along the same slot lanes. For example, reductions that sum many elements within a ciphertext will need to rotate slots so that elements can be summed in one slot. Arbitrary shuffles also have to be implemented using rotates and multiplication with masks, which can require many instructions and quickly become expensive to implement.

Noise. During encryption ciphertexts are injected with random noise to prevent threats such as replay attacks [48]. During computation this noise grows. The ciphertext bitwidth q needs to be large enough to contain this noise growth or else the ciphertext becomes corrupted and upon decryption returns an random value (i.e., garbage value). However, larger values of q increase the memory footprint of ciphertext and requires more compute resource to perform the larger bitwidth arithmetic calculations that back HE instructions.

Specifically, add and rotate additively increase noise, and multiplication multiplicatively increases noise. Because multiplication dominates noise growth, the multiplicative depth of a program can be used as a guide to select q or as a minimization target.

3 HE Compilation Challenges

Handwriting efficient HE kernels is a tedious and error-prone process as HE provides limited instructions, intra-ciphertext data movement must be done using vector rotation, and the noise budget adds additional sources of error. As a result, HE code is today is typically written by experts [16, 27, 40].

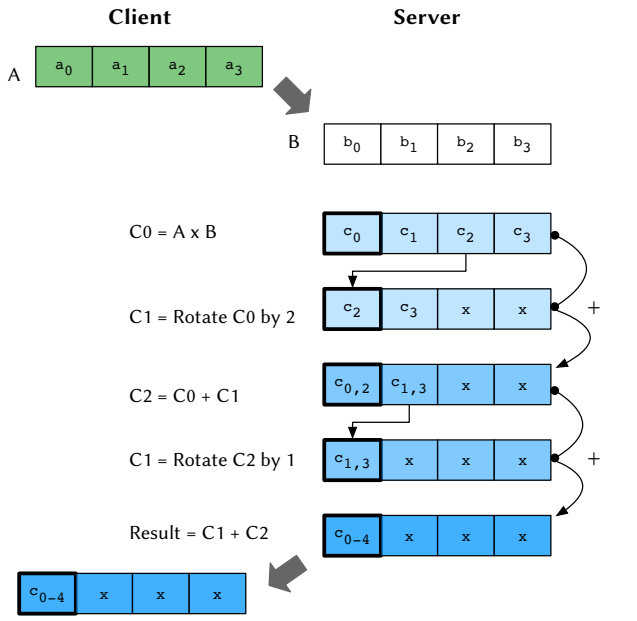

Figure 2. HE vectorized dot product implementation. Given an encrypted input from the client (A), the server performs an element-wise multiplication with server-local data (B). A reduction is performed using a combination of rotation and add instructions. The resulting ciphertext is then returned the client for decryption.

Porcupine's goal is to automate the generation of vectorized HE kernels to lower HE's high barrier-to-entry to non-experts as well as time-to-solution for experts. This section motivates the need for automated reasoning in HE compilers using a vectorized dot product (see Figure 2) as a running example.

3.1 Data Packing

To compute an HE dot product, a client sends an encrypted vector of elements to be computed with a server's vector; the encrypted result is then sent back to the client. A client could encrypt each element of the input vector into individual ciphertexts, but this uses only a single slot of each ciphertext vector, wasting the other slots. Another solution is to batch N independent tasks into a single ciphertext to amortize the cost of the ciphertext and HE program. However, HE vectors can hold tens of thousands of elements and most applications cannot rely on batching of this scale.

Instead, a client can pack the input data vector in a single ciphertext, as shown in Figure 2. In our example of a four element dot product, this requires only one ciphertext, not four. *Porcupine assumes kernels operate over packed inputs to efficiently utilize memory.*

3.2 HE Computation

One of the key challenges for building optimized HE kernels is breaking down scalar computation to efficiently use the limited HE instruction set. In ciphertext vectors, the relative ordering of packed data elements is fixed; thus, while element-wise SIMD addition and multiplication computation is trivial to implement, scalar-output calculations such as reductions require proper alignment of slots between ciphertext operands. The only way to align different slot indices between two ciphertexts is to explicitly rotate one of them such that the desired elements are aligned to the same slot.

Figure 2 illustrates how this is done for an HE dot product reduction operation using packed vectors. The client's and server's ciphertext operands are multiplied together and reduced to a single value. The multiplication operation is element-wise, so it can be implemented with a HE SIMD multiply operation. However, the summation within the vector must be performed by repeatedly rotating and adding ciphertexts together such that the intermediate operands are properly aligned to a slot in the vector (in this case the slot at index 0). The rotations and arithmetic operations are interleaved to take advantage of the SIMD parallelism and enable reduction to be computed with only two HE add operations for four elements.

For more complex kernels, simultaneously scheduling computations and vector rotations is non-trivial to implement efficiently. Arbitrary slot swaps or shuffles (e.g., instructions like `_mm_shuffle_epi32`) that change the relative ordering of elements in a vector are even more tedious to implement. While these arbitrary shuffles can be implemented in HE by multiplying with masks and rotation operations, this is undesirable since it requires dramatically increasing the multiplicative depth and hence noise budget requirements.

3.3 Performance and Noise

The vectorization challenges are further complicated by HE's compound-cost model that must consider both performance and noise. Performance and noise costs cannot be reasoned about independently; the performance cost must be aware of the noise growth since the noise budget parameter q defines the bitwidth precision of the underlying mathematical HE instruction implementations. Thus, a larger q increases the latency cost of each HE instruction. This means any sort of optimization objective for program synthesis will have to consider both noise and performance together.

4 Porcupine Compiler and Synthesis Formulation

This section introduces the Porcupine compiler, Quill DSL, and the program synthesis formulation used to optimize HE kernels.

4.1 Compiler Overview

Porcupine is a program synthesis-based compiler that searches for HE kernels rather than relying on traditional rewrite

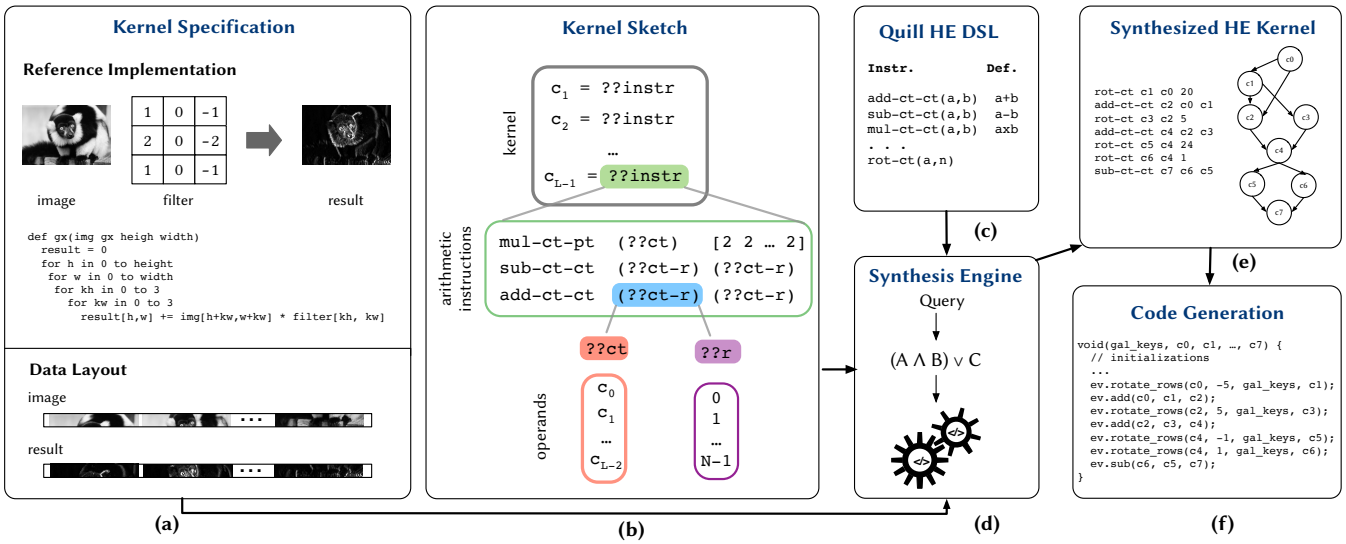

Figure 3. The Porcupine compiler. The user provides: (a) a kernel specification and (b) a kernel sketch with ?? denoting holes in the sketch. (c) The Quill DSL encodes the semantics of the HE instruction set and noise models. (d) Porcupine's synthesis engine completes the sketch and synthesizes a program that implements the specification using the Quill DSL. Porcupine uses an SMT solver to automatically solve the vectorization and scheduling challenges so that (e) the synthesized program is optimized. (f) The optimized HE kernel is consumed by code generation to target the SEAL library [44]. Lemur image ©Skip Brown, Smithsonian's National Zoo.

rules. By searching for programs, Porcupine can discover optimizations that are potentially difficult to identify by hand. At a high level, Porcupine takes a kernel specification (Figure 3a) and sketch (Figure 3b) as input, and outputs an optimized HE kernel (Figure 3f). Section 4.2 defines our Quill DSL (Figure 3c) which is used to model the noise and latency semantics of each HE instruction. Section 4.3 defines what composes the specification that Porcupine as input (Figure 3a). Section 4.4 explains our sketch formulation and design decisions behind them (Figure 3b). Section 5 details our synthesis engine [2] which takes the specification, sketch, and HE kernel, and emits a synthesized HE kernel (Figure 3).

4.2 Quill: A DSL for HE

The Quill DSL serves as a specification for HE programs and models HE ciphertexts, instructions, and their latency-noise behavior. This enables Porcupine to reason about HE instruction behavior as well as verify correctness. When synthesizing an HE kernel, Porcupine first synthesizes a Quill kernel which is then translated into code for an HE library such as SEAL. Quill currently supports BFV [18] HE, however the techniques are general and can be extended to other ring-based HE schemes, e.g., BGV [10] and CKKS [13].

Quill is used to describe straight-line HE programs that manipulate state initially defined by input vectors (either ciphertext and plaintext) and returns a ciphertext. Figure 4 defines Quill's grammar. Quill programs are behavioral models and not true HE programs. The ciphertext operands are implemented as unencrypted vectors that can only be manipulated according to HE instruction rules, which are captured by Quill's semantics. This provides the benefit that we can compile code without considering the implementation details of true HE.

$$
\begin{aligned}
\langle kernel \rangle &::= (\texttt{list}\ \langle instr \rangle +) \mid \langle instr \rangle \\
\langle instr \rangle &::= \langle ct \rangle \\
\langle pt \rangle &::= \langle vector\ of\ integers \rangle \\
\langle x \rangle &::= \langle integer \rangle \\
\langle ct \rangle &::= (\texttt{add-ct-ct}\ \langle ct \rangle, \langle ct \rangle) \mid (\texttt{add-ct-pt}\ \langle ct \rangle, \langle pt \rangle) \\
&\quad \mid (\texttt{sub-ct-ct}\ \langle ct \rangle, \langle ct \rangle) \mid (\texttt{sub-ct-pt}\ \langle ct \rangle, \langle pt \rangle) \\
&\quad \mid (\texttt{mul-ct-ct}\ \langle ct \rangle, \langle ct \rangle) \mid (\texttt{mul-ct-pt}\ \langle ct \rangle, \langle pt \rangle) \\
&\quad \mid (\texttt{rot-ct}\ \langle ct \rangle, \langle x \rangle)
\end{aligned}
$$

Figure 4. The Quill DSL. A Quill kernel is made up of a list instructions that produce a final ciphertext (ct). Each instruction produces a ciphertext and takes as input at least one ciphertext and possibly a plaintext (pt) or integer.

State in Quill. In a Quill program, state is defined by plaintext and ciphertext vectors. All ciphertexts are associated with metadata that tracks each operand's multiplicative depth, which models noise accumulation. An input or fresh ciphertext has zero multiplicative depth and increases each

time a multiplication is performed. We track only multiplicative depth as it simplifies the objective of noise minimization without sacrificing accuracy as other instructions - add and rotate - contribute relatively negligible noise.

The Quill Instructions. Quill supports a SIMD instruction set with a one-to-one mapping to BFV HE instructions. Table 1 describes each instruction's type signature and how they transform state. Instructions include addition, multiplication, and rotations of ciphertext instructions as well as variants that operate on ciphertext-plaintext operands, e.g., multiplication between a ciphertext and plaintext. Each instruction is associated with a latency derived by profiling its corresponding HE instruction with the SEAL HE library [44].

4.3 Kernel Specification

A *specification* completely describes a target kernel's functional behavior, i.e., it defines what the synthesized HE kernel must compute. In Porcupine, a specification comprises a *reference implementation* of the computation (in plaintext) and *vector data layout* that inputs and outputs must adhere to.

Reference Implementation. Reference implementations are programs written in Racket [38] that define the plaintext computation. We later use Rosette [49] to automatically lift the Racket program to a symbolic input-output expression that defines the program's behavior. An example reference implementation for the G_x kernel is shown below. The code takes as input a 2D gray-scale image and calculates the x-gradient by summing and weighting neighboring pixels according to a 3×3 filter.

```
(define (Gx img height width filter):
    for h in 0 to height
        for w in 0 to width:
            for kh in 0 to 3:
                for kw in 0 to 3:
                    result[h,w] += img[h+kw, w+kw] *
                                   filter[kh, kw]
```

Porcupine uses the reference implementation to verify synthesized ones are correct; the quality of the reference program does not impact synthesized code quality. As a result, users can focus on writing correct code without the burden of performance tuning.

To work correctly, the implementation must describe computation that is directly implementable in HE. Implementations cannot contain data dependent control flow such as conditional statements or loops that depend on a ciphertext, since we cannot see the values of encrypted data. This is a limitation of HE, and while it is possible to approximate this behavior, e.g., using a polynomial function, this is beyond the scope of our work.

Data Layout. A data layout defines how the inputs and outputs are packed into ciphertext and plaintext vectors. In the G_x example, we pack the input and output image into one ciphertext as a flattened row-order vector with zero-padding around the borders. The data layout is an input to the synthesizer only, and the reference implementation does not need to consider it. Together, the reference implementation and data layout define the inputs and outputs to the HE program, and Porcupine will synthesize an HE program that achieves that transformation.

4.4 Sketch

The user also provides a *sketch*, which is a template for describing partial Quill kernels that are used to guide the synthesis engine towards a solution. It allows the user to articulate required features of the HE kernel to the synthesis engine while leaving other components unspecified as *holes*, indicated by ??, for the engine to fill in. The synthesizer then completes the sketch by filling in the holes to match the functionality of the reference implementation. We introduce a local rotate sketch to help the user convey hints about ciphertext rotations. An example of a local rotate sketch for the G_x kernel is shown below:

```
; Program sketch of L components
; ct0 is a ciphertext input
(define (Gx-Sketch ct0 L)
    ; choose an existing ciphertext
    (define (??ct)  (choose* ct))
    ; choose a rotation amount in range (0,N)
    (define (??r)
      (apply choose* (range 0 N)))
    ; choose a rotation of an existing ciphertext
    (define (??ct-r)
        (rot-ct ??ct ??r))
    ; choose an opcode with operand holes
    (for/list i = 1 to L
        (choose*
            (add-ct-ct (??ct-r) (??ct-r))
            (sub-ct-ct (??ct-r) (??ct-r))
            (mul-ct-pt (??ct) [2 2 ... 2]))))
```

The sketch describes a kernel template that takes as input a single ciphertext (encrypted image) and applies a kernel composed of `L` *components* or arithmetic instructions. In this example the components are: add two ciphertexts, subtract two ciphertexts or multiply a ciphertext by a plaintext of 2s. Each component contains holes for their instruction dependent operands. Specifically, `??ct` is ciphertext hole that can be filled with the ciphertext input or a ciphertexts generated by previous components. `??ct-r` is a ciphertext-rotation that introduces two holes: a ciphertext hole and a rotation hole. The ciphertext hole can be filled with any previously generated ciphertexts and the rotation hole indicates the ciphertext can be rotated by any legal amount (1 to $N-1$) or not at all. Ciphertext-rotation holes indicate the kernel performs a reduction operation over elements and requires rotation to align vector slots.

Writing sketches of this style is relatively simple, with most sketches taking only a few minutes to write and debug. The arithmetic instructions can be extracted from the specification. In this case add, subtract, and multiplication by 2 were used in the reference implementation. The set of arithmetic instructions is treated like a multiset of multiplicity L, and

Table 1. Quill instructions and their affect on the data (denoted by *.data*) and multiplicative depth (denoted by *.depth*) of the resulting ciphertext.

Instruction	Computation	Description	Multiplicative depth
Add(ct_x, ct_y) $\rightarrow ct_z$	$ct_x.data + ct_y.data$	Adds two ciphertexts	$max(ct_x.depth, ct_y.depth)$
Add(ct, pt) $\rightarrow ct_z$	$ct.data + pt.data$	Adds a ciphertext and plaintext	$ct.depth$
Subtract(ct_x, ct_y) $\rightarrow ct_z$	$ct_x.data - ct_y.data$	Subtract two ciphertexts	$max(ct_x.depth + ct_y.depth)$
Subtract(ct, pt) $\rightarrow ct_z$	$ct.data - pt.data$	Subtract a plaintext from a ciphertext	$ct.depth$
Multiply(ct_x, ct_y) $\rightarrow ct_z$	$ct_x.data \times ct_y.data$	Multiple two ciphertexts	$max(ct_x.depth, ct_y.depth) + 1$
Multiply(ct, pt) $\rightarrow ct_z$	$ct.data \times pt.data$	Multiply a ciphertext and plaintext	$ct_x.depth + 1$
Rotate(ct, x) $\rightarrow ct_z$	$ct.data[i] \leftarrow ct.data[(i+x) mod N]$	Rotate a ciphertext x slots to the left	$ct.depth$

the synthesizer will determine which instructions and how many are needed. In other words, the sketch does not have to be exact as the synthesizer can choose to ignore instructions; this once again eases the burden on the user. Additionally, the user must specify whether instruction operands should be ciphertexts or ciphertext-rotations, and what rotations are allowed. As a fall back, all ciphertext holes can be made ciphertext-rotation holes; however, this will increase solving time as the sketch describes a larger space of programs. Furthermore, the effort of sketch writing can potentially be amortized by re-using or tweaking a sketch from a kernel with similar compute patterns. For example, when writing a sketch for a different 2D convolution, we could start from this `Gx-sketch` and either re-use it or change the plaintext constants.

A key feature of our sketches is that we treat rotation as an input to arithmetic instructions rather than a component of the sketch. This is because rotations are only useful when an arithmetic instruction needs to re-align operands; in isolation, rotations do not perform meaningful computation. This excludes programs that contain nested rotations since rotations can be combined. For instance, we disallow (rot (rot c0 1) 2) since this can be more succinctly expressed as (rot c0 3).

The sketches must describe loop-free programs so that Quill can interpret them. Porcupine requires sketches to be parameterized by the number of components in the program. Porcupine first explores small (in terms of L) programs and iteratively explores larger programs by incrementing L until a solution is found.

Solution. A *solution* is a completed sketch that matches the behavior of the reference implementation. Porcupine's synthesis engine generates solutions by filling instruction and operand holes such that the resulting program satisfies the specification and optimizes the objective functions (minimize instruction count and noise). The solution Porcupine synthesizes for the above example uses three arithmetic instructions and four rotations [1]:

```
Ciphertext c1 = (add-ct-ct (rot-ct c0 -5) c0)
Ciphertext c2 = (add-ct-ct (rot-ct c1 5) c1)
Ciphertext c3 = (sub-ct-ct (rot-ct c2 1)
                           (rot-ct c2 -1))
```

5 Synthesis Engine

This section describes how Porcupine's synthesis engine (see Algorithm 1) searches the program space described by our local rotate sketch to find an optimized HE solution that satisfies the kernel specification. Porcupine's synthesis engine operates by first synthesizing an initial solution. It then optimizes the solution by iteratively searching for better solutions until either the best program in the sketch is found or a user-specified time out is reached.

Porcupine's synthesis engine is a counter-example guided inductive synthesis (CEGIS) loop [26, 46]. The engine leverages Rosette's built-in support for translating synthesis and verification queries to constraints that are solved by an SMT solver.

5.1 Synthesizing an Initial Solution

The first step in Porcupine's synthesis procedure is to synthesize an initial program that satisfies the user's specification. In particular, Porcupine first attempts to complete a sketch $sketch_L$ that encodes programs using L components. Specifically, Porcupine searches for a solution sol_0 contained in $sketch_L$ that minimizes L and satisfies the specification for all inputs. We follow a synthesis procedure similar to those proposed in [23, 26, 46], and avoid directly solving the above query because it contains a universal quantifier over inputs. Instead, we synthesize a solution that is correct for one random input then verify it is correct for all inputs, applying feedback to the synthesis query if verification fails.

[1] Rotation amounts are adjusted to be relative in example.

Synthesize. The engine starts by generating a concrete input-output example, (x_0, y_0), by evaluating the specification using a randomly generated input, x_0 (line 6). The engine attempts to synthesize a program that transforms x_0 into y_0 by completing the sketch and finding a binding for the L arithmetic instructions and operand holes (line 10). We generate a synthesis query expressing $solve(sketch_L(x_0) = y_0)$, which is then compiled to constraints and solved by an SMT solver.

Verify. If successful, the synthesis query described above returns a program that satisfies the input specification for the input x_0, but not necessarily for all possible inputs. To guarantee that the solution is correct, Porcupine verifies the solution matches the specification for all inputs. Porcupine leverages Rosette's symbolic evaluation and verification capabilities to solve this query. First, a formal pre-condition and post-condition is lifted from reference specification with symbolic execution, capturing the kernel's output for a bounded set of inputs as a symbolic input-output pair $(\hat{x}, \hat{y})$. Rosette then solves the verification query $verfiy(sol(\hat{x}) = spec(\hat{x}))$.

Retry with Counter-example. If verification fails, it returns a counter-example, (x_1, y_1), that causes the synthesized kernel to disagree with the specification. Porcupine then makes another attempt to synthesize a program; this time trying to satisfy both the initial example and counter-example. This process repeats until Porcupine finds a correct solution.

If the engine cannot find a solution, indicated when the solver returns `unsat` for any synthesis query, the engine concludes that for the given sketch, a program that implements the specification with L components does not exist. The engine tries again with a larger sketch $sketch_{L+1}$ that contains one more component and this process repeats until a solution is found. By exploring smaller sketches first, our algorithm ensures that the solution using the smallest number of components is found first.

5.2 Optimization

Once an initial solution is found, Porcupine's synthesis engine attempts to improve performance by searching for better programs contained in the sketch. Programs are ranked according to a cost function that Porcupine attempts to minimize.

Cost Function. Porcupine uses a cost function that multiplies the estimated latency and multiplicative depth of the program: $cost(p) = latency(p) \times (1 + mdepth(p))$. We include multiplicative depth to penalize high-noise programs, which can lead to larger HE parameters and lower performance.

Cost Minimization. Once a solution sol_0 with cost $cost_0$ is found, we iteratively search for a new program with lower cost (line 19), as described in [45]. Porcupine does this by re-issuing the successful synthesize query with an additional constraint that ensures a new solution sol_1, has lower cost: $cost_1 < cost_0$ (line 25). This process repeats until the solver proves there is no lower cost solution and it has found the best solution or the compile time exceeds the user-specified time out. The initial solution is only used to provide an upper-bound on cost and is not used during the optimization synthesis queries. This forces the engine to consider completely new programs with different instruction mixes and orderings. In practice, we find that initial solutions perform well given the savings in compile time (see Section 7.4 for discussion).

Algorithm 1 Synthesis engine

```
1:  Input
2:      spec        Kernel reference program
3:      sketch      Partial HE program
4:  Synthesize first solution
5:  function SYNTHESIZE
6:      y_0 ← spec(x_0)            ▷ Random input-output example
7:      ŷ = spec(x̂)                ▷ Symbolic input-output
8:      examples = [(x_0, y_0)]
9:      while true do
10:         sol ← solve(sketch s.t. y=sketch(x))
11:         if sol is unsat then
12:             return False        ▷ Sketch too restrictive
13:         cex ← verify(ŷ = solution(x̂))
14:         if cex = unsat then
15:             return sol
16:         (x, y) ← extract(cex)   ▷ Get counterexample
17:         examples.append((x, y))
18: Minimize cost
19: function OPTIMIZE
20:     sol ← synthesize()
21:     c' ← cost(sketch)
22:     sol' ← sol
23:     while sol' is sat do
24:         c ← cost(sol), sol ← sol'
25:         sol' ← solve(sketch s.t. y=sketch(x) & c' < c)
26:         <verify sol' and add cex if needed>
27:     return sol
```

5.3 Code Generation

The synthesis engine outputs a HE kernel described in Quill and Porcupine then translates the Quill program into a SEAL program [44]. SEAL is a HE library that implements the BFV scheme. Quill instructions map directly to SEAL instructions, so this translation is simple, but the code generation handles a few post-processing steps. For example, Porcupine inserts special *relinearization* instructions after each ciphertext-ciphertext multiplication. Relinearization does not affect the results of the HE program but is necessary to handle ciphertext multiply complexities.

6 Synthesis Formulation Optimizations

Scaling Porcupine to handle larger kernels requires optimizing the synthesis formulation. Since the search space grows super exponentially, it quickly becomes intractable—a five instruction HE program can have millions of candidate programs. This section describes optimizations developed to scale up our formulation and their impact on the results.

6.1 Rotation Restrictions

HE Rotation instructions are used to align different vector slots within a ciphertext to perform computation such as reductions. Ciphertext slots can be rotated by up to N, the size of the ciphertext vector, which introduces a large number of possible rotations for the synthesizer to select from. In practice, we observe that of all possible rotations only a few patterns are ever used. For example, in our G_x kernel each output element only depends on its neighbors in the 3×3 window, implying rotations that align input elements from outside this window are not necessary. By restricting rotations, we can scale up the synthesis process by pruning away irrelevant potential programs.

To optimize for this, we introduce two types of rotation restrictions for tree reductions and sliding windows. For sliding window kernels, which are commonly used in image processing, we use the restriction described above to restrict rotation holes `??rot` to align elements covered by the window. The tree reduction restricts rotations to powers of two and is used for kernels that implement an internal reduction within the ciphertext. For example, in a dot product elements in the vector are summed to produce one value. Restricting the rotations to powers of two constrains the output programs to perform the summation as a reduction tree.

6.2 Constraint Optimizations

We also apply a number of common constraint optimization techniques to improve synthesis speed and scalability. We employ symmetry breaking to reduce the search space for add, multiply, and rotate. For example, the programs $a + b$ and $b + a$ are functionally equivalent but appear as two unique solutions to a solver. Restricting operands to occur in increasing order eliminates redundant candidate solutions and improves synthesis speed. For rotations we impose symmetry breaking by forcing only left rotations, since a left rotation by x is equivalent to a right rotation by $n - x$. We also enforce solutions use static single assignment to instill an ordering and break symmetries between programs that are functionally equivalent but write to different destination ciphertexts.

Our synthesis formulation also uses restricted bitwidth instead of full precision bit vectors to reduce the number of underlying variables the solver needs to reason about. Ordinarily, the number of solver variables scales linearly with bitwidth; however, we do not need the bit accurate behavior, only the operator functionality, so this optimization does not affect correctness of the solution.

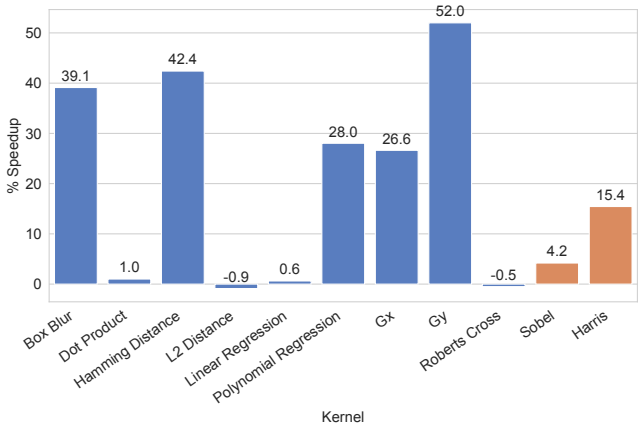

Figure 5. Speedup of Porcupine synthesized kernels compared to the baseline, results are averaged over 50 runs. Kernels in blue are directly synthesized while kernels in orange use multi-step synthesis.

6.3 Multi-step Synthesis

One of the limitations of program synthesis is its inability to scale to large kernels [24]. With the above optimizations, Porcupine scales to roughly 10-12 instructions, but beyond that the program space becomes intractable to search. Many applications in image processing, neural networks, and machine learning have natural break points. For instance, an image processing pipeline may have cascaded stencil computations for sharpening, blurring, and edge detection which have natural boundaries. To scale beyond the limitations of program synthesis, we leverage these natural breakpoints to partition larger programs into segments and synthesize them independently. In Section 7, we show how this partitioning into a multistep synthesis problem can allow Porcupine to scale to longer kernels.

7 Evaluation

This section evaluates Porcupine's synthesized programs and compares them against expert-optimized baselines (see Section 7.2). We also report how long Porcupine takes to synthesize kernels (see Section 7.4). We find that Porcupine is able to synthesize a variety of kernels that are at least as good or better than an expert-written version, and in most cases can synthesize a kernel in a few minutes.

7.1 Methodology

Porcupine is implemented with Rosette v3.1 [49], and configured to use Boolector [11] as its backend SMT solver. Synthesized kernels are compiled down to SEAL v3.5's BFV library [44]. When running Porcupine's kernels, security

Table 2. A comparison of instruction count, multiplicative depth (M. Depth), and logical depth (L. Depth) of synthesized and baseline kernels.

Kernel	Synthesized/Baseline		
	Instr.	M. Depth	L. Depth
Box Blur	4/6	0/0	4/3
Dot Product	7/7	1/1	7/7
Hamming Distance	9/13	1/1	9/9
L2 Distance	7/7	1/1	7/7
Linear Regression	4/4	1/1	4/4
Polynomial Regression	7/9	2/2	5/5
Gx	7/12	0/0	4/4
Gy	7/12	0/0	4/4
Roberts Cross	10/10	1/1	5/5
Sobel	19/25	1/1	9/7
Harris	43/59	3/3	17/14

parameters are set to guarantee a 128-bit security level; both baseline and synthesized kernels use the same settings. All experiments are conducted on a 3.7 GHz Intel Xeon W-2135 CPU with 64 GB of memory.

Workloads. We evaluate Porcupine using common kernels found in linear algebra, machine learning, and image processing listed in Table 3. Since there is no standardized benchmark for compiling HE kernels, we attempt to be as diverse and representative in our selection as possible. For example, dot product, L2 distance, and linear and polynomial regression kernels are building blocks of machine learning applications, while the x/y-gradient (G_x/G_y) and Roberts Cross kernels are used in image processing applications.

Kernels are modified to omit operations not directly supported by HE. For instance, the canonical L2 distance kernel uses a square root, but many applications (e.g., k-nearest neighbors) can use squared distance with negligible effect on accuracy [31]. Finally, because BFV cannot directly implement data-dependent branches or conditionals, applications that require these operations are calculated up to a branch. For example, our Harris corner detector implementation returns an image of response values that the user must decrypt and apply a threshold over to detect the corners.

Baselines. We compare Porcupine's code quality against an expert's hand-written implementation that seeks to first minimize multiplicative, then logical depth. Minimizing multiplicative depth was chosen to reflect the state-of-the-art solution that was recently proposed for optimizing HE kernels under Boolean HE schemes [28]. The paper suggests that optimizing multiplicative depth also minimizes noise, as fewer successive operations compound less noise between any input-output. Since some of our baseline kernels require few or no multiplications, the baselines further minimize noise growth by minimizing logical depth after multiplicative depth. To minimize depth, these programs attempt to perform as much computation as possible in early levels of the program and implement all reductions as balanced trees. In addition, all our baseline implementations use packed inputs (i.e., are not scalar implementations) to minimize latency.

7.2 Synthesized Kernel Quality

To understand the quality of Porcupine's synthesized programs, we compare instruction count, multiplicative depth, logical depth, and run time against the hand-optimized baseline. We report run time speedups in Figure 5, with all times averaged over 50 independent runs and instruction counts in Table 2.

The results show that Porcupine's kernels have *similar or better performance* compared to the hand-written baselines. For some kernels such as dot product, L2 distance, and Roberts Cross, Porcupine generates roughly the same kernel as the hand-written implementation. The synthesized and baseline implementations may have different orderings of independent instructions, resulting in small performance differences.

For more complex kernels (G_x, G_y), polynomial regression, and box blur), we observe Porcupine's programs have notably better run times, up to 52% and use fewer instructions. Our speedups are a result of Porcupine being able to identify different types of optimizations. For example, our synthesized polynomial regression kernel found an algebraic optimization that factored out a multiplication similar to $ax^2 + bx = (ax + b)x$, resulting in a kernel that used 7 instructions instead of 9 and was 28% faster than the baseline. We analyze more of these optimizations in Section 7.3.

For these kernels, each handwritten baseline took on the order of a few hours to a day to implement, debug, and verify; for a non-expert unfamiliar with HE and SEAL, this would take much longer. The results show that Porcupine can effectively automate the tedious, time-consuming task of handwriting these kernels without sacrificing quality.

Multi-step Synthesis Evaluation. We also used Porcupine's synthesized kernels to compile larger HE applications. Specifically, Porcupine's G_x and G_y kernels are used to implement the Sobel operator, and G_x, G_y, and box blur kernels were used to implement the Harris corner detector, shown in orange in Figure 5. By leveraging Porcupine synthesized kernels, our Sobel operator and Harris corner detector were 4% and 15% faster than the baseline, and used 10 and 16 fewer instructions respectively. These results show that we can speedup larger applications by synthesizing the core computational kernels these applications rely on.

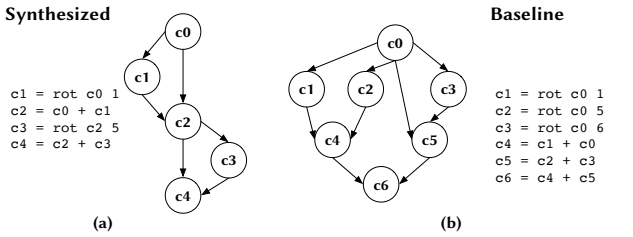

Figure 6. HE kernels for box blur. (a) Synthesized kernel with minimal number of instruction (b) Hand-optimized minimal depth kernel. Porcupine achieves a much higher performing kernel by separating kernels and use fewer instructions which, even though the logical depth increases, results in a 39% speedup.

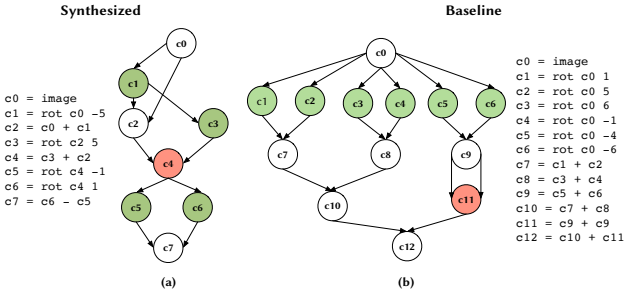

Figure 7. (a) Synthesized and (b) baseline G_x kernel. The synthesized kernel uses 7 instructions while the baseline uses 12 instructions. The synthesized kernel optimizes the computation to separate the 2D convolution into two 1D convolutions and interleaves rotation and computation. Ciphertexts generated by rotations are marked in green and the ciphertext where multiplication by 2 is implemented with an addition is in red.

7.3 Analysis of Synthesized Kernels

We now analyze the synthesized and baseline implementations of the box blur and G_x kernels to demonstrate the tradeoffs explored by Porcupine. Figure 6 compares Porcupine's and the baseline's box blur. The baseline implements this kernel in six instructions with three levels of computation. In the first level, elements are aligned in the window with rotations and then summed in a reduction tree. Porcupine's synthesized kernel uses four instructions with five levels; decomposing the 2D convolution into two 1D convolutions to perform the same computation with fewer instructions. Furthermore, despite having a greater logical depth, the synthesized solution consumes the same amount of noise as the baseline. By focusing on minimizing logical depth, the baseline misses the separable kernel optimization because it was not the minimum depth solution.

We observe similar results for the G_x kernel and show the synthesized and baseline programs in Figure 7. The depth-optimized baseline applies the same strategy as the box blur kernel, first aligning elements in the sliding window then combining them in a balanced reduction tree. The G_x kernel weights some of the neighbor elements by two, and the baseline substitutes the multiplication with a cheaper addition (operand c11 in Figure 7b). The synthesized G_x kernel has a very different program structure from the baseline. Porcupine discovers the filter is separable and decomposes the kernel into two 1D filters, requiring a different set of rotations and schedule to implement correctly as depicted in Figure 8. Porcupine's synthesized solutions automatically also substitutes the multiplication by two with an addition which is performed at c4 in Figure 8 in parallel with other additions.

While minimizing for logical depth is a good guideline for minimizing noise in scalar HE programs, our results show it is not guaranteed to find the optimal implementations for vector HE constructions, like BFV, and can leave significant

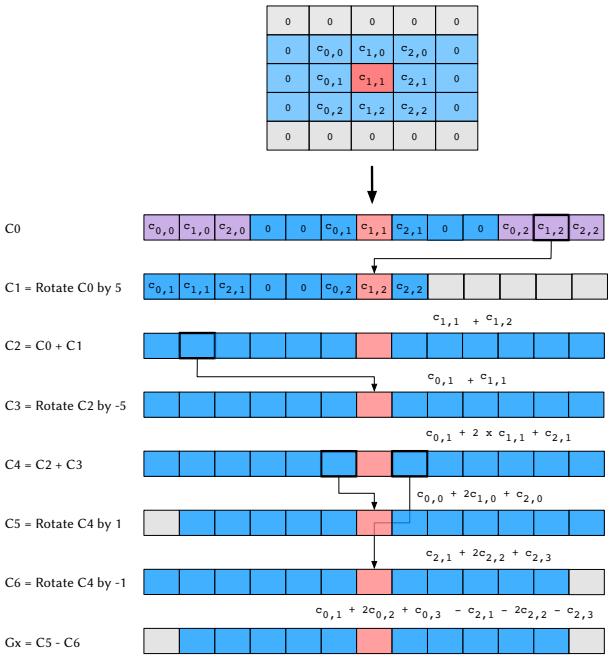

Figure 8. Porcupine optimized G_x kernel. An optimized implementation packs the entire image into one ciphertext and schedules computation with rotations. Purple slots contain elements that are used to compute the final red slot. The value contained in the red slot are tracked on the right hand side.

unrealized performance (e.g., up to 52% for G_y). Because Porcupine searches for vectorized implementations, and tracks program latency and multiplicative depth it can outperform the heuristic for programs with more complex dependencies.

Table 3. Synthesis time and number of examples used by Porcupine. Initial time is the time to synthesize a solution and total time includes time spent optimizing. Reported values come from the median of three runs.

Kernel	Examples	Initial Time (s)	Total Time (s)	Initial Cost	Final Cost
Box Blur	1	1.99	9.88	1182	592
Dot Product	2	1.27	15.16	1466	1466
Hamming Distance	10	38.98	77.21	3428	1658
L2 Distance	2	27.57	114.28	1436	1436
Linear Regression	2	0.50	0.69	878	878
Polynomial Regression	2	24.59	47.88	2631	2631
Gx	1	14.87	70.08	1357	975
Gy	1	9.74	49.52	1773	767
Roberts Cross	1	212.52	609.64	2692	2692

7.4 Synthesis Time

Table 3 reports the median time it took to synthesize each kernel over three runs. We report how long it took to find an initial solution and the cumulative time it took to find an optimized solution. For most of the kernels we were able to synthesize an initial solution in under 30 seconds and synthesize an optimized solution under 2 minutes. The Roberts Cross kernel required more time, taking over 2 minutes to synthesize an initial solution and in total to 27 minutes to optimize. This is because the Roberts Cross kernel required a sketch with 10 instructions, which took longer to search over. Additionally, the optimization phase of the synthesis engine must prove it found the best solution contained in the sketch, requiring the SMT solver explore the entire search space.

In terms of input-output examples required by Porcupine during the synthesis process, we typically only require one example to synthesize a solution; however, for some kernels such as Hamming distance we required up to 10 input-output examples be generated during synthesis. We find kernels that produce a single-valued output, like Hamming distance, require more examples than kernels that produce a vector output (e.g., image processing kernels). This is because the synthesis engine can find many input-dependent (but not general) programs.

Cost Trajectory. Table 3 also reports the cost of the initial and final solutions found by Porcupine. For some kernels, the initial and first solution Porcupine finds are the same. This indicates that either there was only one correct solution in the minimum L-sized sketch, or that Porcupine found the best solution on the first try. The time between Porcupine reporting the initial and final solution is spent proving that it found the best solution in the sketch. After the initial solution is found, users can terminate Porcupine early to shorten compile times. While this does not guarantee the best solution was found, it will minimize arithmetic instructions.

Local Rotate Sketch Analysis. Our local rotate sketches treat rotations as operands instead of components. We could have alternatively required users explicitly add rotations to the list of components supplied in the sketch (which we refer to as *explicit rotation* sketches). However, explicit rotation sketches describe a larger space of programs that includes the space described by our local rotate sketches.

In small kernels, e.g., box blur, the synthesis time using local rotate sketches was slower than the explicit rotation sketch; the explicit rotation sketch took only 3 seconds to synthesize verses 10 seconds when using a local rotate sketch. However, when we synthesize larger programs the explicit rotation sketch scales poorly. Using the explicit rotation sketch, synthesizing the G_x kernel took over 400 seconds to find an initial solution then over 30 minutes total. On the other hand, the local rotate sketches found the same solution in about 70 seconds total, showing that local rotate does improve synthesis scalability and search time.

8 Related Work

8.1 Compilers for Homomorphic Encryption

Recent work proposes domain-specific and general compilers for HE [3, 7, 12–16]. Prior work such as CHET [16] and nGraph-HE [7] are domain-specific HE compilers for deep neural networks (DNNs). CHET optimizes the data layout of operations in DNNs while nGraph-HE added an HE extension to an existing DNN compiler with new graph-level optimizations. Cingulata [12] and Lobster [28] target Boolean HE schemes and propose compilation strategies that rely on multiplicative depth minimization and synthesizing rewrite rules.

Other HE compilers such as EVA [15] and Alchemy [14] automate parameter selection and placement of low-level scheme specific HE instructions that control ciphertext properties necessary for correctness, but have no affect on the result of computation (e.g., mod-switch). For example, EVA achieves this for the CKKS scheme using custom rewrite

rules but requires a hand-crafted HE kernel as input. On the other hand, Porcupine tackles an orthogonal problem of synthesizing vectorized kernels and optimizes the computational instructions.

The closest work to ours is Ramparts [3] which is a HE compiler that translates plaintext Julia programs to equivalent HE implementations. Unlike Porcupine, Ramparts does not support packed vectorization (i.e., one task cannot use multiple slots in a ciphertext) which is required for taking advantage of SIMD parallelism within a task and improving latency. In contrast, Porcupine supports packed data inputs and can generate kernels with rotations. Furthermore, Ramparts relies on the structure of the input Julia program to serve as the seed for symbolic execution-based methodology which produces a computational circuit that is optimized and lowered to HE instruction with rewrite rules. In contrast, Porcupine places essentially no constraints on the structure of the programs it synthesizes other than the number of instructions it can contain. This enables Porcupine to consider a wider range of programs when optimizing.

Overall, Porcupine is the first compiler that applies program synthesis to optimize vectorization for integer HE constructions.

8.2 Compilers for Privacy-Preserving Computation

Compiler support has also been proposed for other privacy-preserving techniques, such as differential privacy (DP) [17] and secure multi-party computation (MPC) [22, 51] to automatically enforce or reason about restrictions and constraints by these technologies. For instance, DP requires adding noise to the algorithm and deriving that the effect of an individual's information is in fact differentially private (i.e., has indistinguishable effect on the aggregate data). In DP, there are proposals for using type systems to enforce differential privacy [19, 32, 41]. Other programming language techniques [5] include dynamic approaches [29, 30, 43], static checking [19, 33, 41], and machine-checked proofs [6]. A similar trend is occurring in MPC where implementations must also comply with design constraints to collaboratively compute functions while still protecting private inputs from other users. Recent work by [25, 39, 47, 50, 52] proposes and/or evaluates general-purpose compiler for MPC.

8.3 Synthesizing Kernels

Prior work has shown program synthesis to be effective for compiling and optimizing programs for various goals and targets. For example, Chlorophyll [35] introduced a synthesis-based compiler that targets a scalar spatial architecture with a stack-based language. By pairing a naive code generator with a synthesis based superoptimizer they were able to quickly build an optimizing compiler. Spiral [37] generates optimized DSP kernels using both inductive and deductive synthesis techniques

Swizzle Inventor [34] synthesized optimized data movement for GPU kernels from a sketch that specified that computation strategy and left data movement unspecified. Because their objective only optimized data movement, they relied on canonicalization for verification (not an SMT solver) which does not allow their synthesis formulation to optimize algebraic expressions but improves synthesis time. On the other hand, our synthesis formulation needs to optimize algebraic expressions as part of selecting arithmetic instructions so requires an SMT solver.

Program synthesis has also been used for automatically vectorizing code. For example, Barthe et al. introduced an auto-vectorization method [4] that transformed scalar loops into SIMD implementations (Intel SSE4) by restructuring loops to expose parallelism and then synthesizing a straight-line SIMD loop body using an enumerative search.Porcupine does not rely on a loop restructuring phase and our synthesis procedure optimizes the entire kernel, allowing us to handle nested loops. Furthermore, our search optimizes an HE cost model that accounts for multiplicative depth and handles vectors larger than four lane CPU SIMD vectors.

9 Conclusion

We presented Porcupine, a program synthesis-based compiler that automatically generates vectorized HE kernels. Porcupine automatically performs the instruction selection and scheduling to generate efficient HE kernels and minimize the HE noise budget. By automating these tasks, Porcupine abstracts away the details of constructing correct HE computation so that application designers can concentrate on other design considerations. HE is still a rapidly maturing area of research and there is limited related work in this space. As a result, we expect that in future work we will see rapid improvements to compilation infrastructure such as ours.

Acknowledgements

We thank our shepherd Woosuk Lee and the anonymous reviewers for their helpful feedback. Additionally, we thank our colleagues James Bornholt, Emily Furst, Liang Luo, and Amrita Mazumdar for their suggestions and advice.

References

[1] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. 2018. *Homomorphic Encryption Security Standard.* Technical Report. HomomorphicEncryption.org, Toronto, Canada.

[2] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. 2013. *Syntax-guided synthesis.* https://doi.org/10.1109/FMCAD.2013.6679385

[3] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. 2019.

RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC'19)*. https://doi.org/10.1145/3338469.3358945

[4] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. 2013. From Relational Verification to SIMD Loop Synthesis. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. https://doi.org/10.1145/2442516.2442529

[5] Gilles Barthe, Marco Gaboardi, Justin Hsu, and Benjamin Pierce. 2016. Programming Language Techniques for Differential Privacy. *ACM SIGLOG News* (2016). https://doi.org/10.1145/2893582.2893591

[6] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic Relational Reasoning for Differential Privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. https://doi.org/10.1145/2103656.2103670

[7] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. 2019. NGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC'19)*. https://doi.org/10.1145/3338469.3358944

[8] Joppe W Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. 2013. Improved security for a ring-based fully homomorphic encryption scheme. In *IMA International Conference on Cryptography and Coding*. Springer, 45–64.

[9] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Advances in cryptology–crypto 2012*. Springer.

[10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Trans. Comput. Theory* (2014). https://doi.org/10.1145/2633600

[11] Robert Brummayer and Armin Biere. 2009. Boolector: An efficient SMT solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 174–177.

[12] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. 2015. Armadillo: a compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*. 13–19.

[13] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 409–437.

[14] Eric Crockett, Chris Peikert, and Chad Sharp. 2018. ALCHEMY: A Language and Compiler for Homomorphic Encryption Made EasY. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. https://doi.org/10.1145/3243734.3243828

[15] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. https://doi.org/10.1145/3385412.3386023

[16] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*. https://doi.org/10.1145/3314221.3314628

[17] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*. Springer, 265–284.

[18] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptol. ePrint Arch.* 2012 (2012), 144.

[19] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. 2013. Linear Dependent Types for Differential Privacy. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. https://doi.org/10.1145/2429069.2429113

[20] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing (STOC '09)*. https://doi.org/10.1145/1536414.1536440

[21] Craig Gentry. 2010. Computing Arbitrary Functions of Encrypted Data. *Commun. ACM* (2010). https://doi.org/10.1145/1666420.1666444

[22] Oded Goldreich, Silvio Micali, and Avi Wigderson. 2019. How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 307–328.

[23] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. https://doi.org/10.1145/1993498.1993506

[24] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* (2017).

[25] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. 2019. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *2019 IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP.2019.00028

[26] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*. https://doi.org/10.1145/1806799.1806833

[27] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. Gazelle: A low latency framework for secure neural network inference. *arXiv preprint arXiv:1801.05507* (2018).

[28] DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 2020. Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. https://doi.org/10.1145/3385412.3385996

[29] Frank McSherry and Ratul Mahajan. 2010. Differentially-Private Network Trace Analysis. In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)*. https://doi.org/10.1145/1851182.1851199

[30] Frank D. McSherry. 2009. Privacy Integrated Queries: An Extensible Platform for Privacy-Preserving Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. https://doi.org/10.1145/1559845.1559850

[31] Marius Muja and David Lowe. 2009. Flann-fast library for approximate nearest neighbors user manual. *Computer Science Department, University of British Columbia, Vancouver, BC, Canada* (2009).

[32] Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, and Dawn Song. 2019. Duet: An Expressive Higher-Order Language and Linear Type System for Statically Enforcing Differential Privacy. *Proc. ACM Program. Lang.* OOPSLA (2019). https://doi.org/10.1145/3360598

[33] Catuscia Palamidessi and Marco Stronati. 2012. Differential privacy for relational algebra: Improving the sensitivity bounds via constraint systems. *arXiv preprint arXiv:1207.0872* (2012).

[34] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. 2019. 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 (ASPLOS '19)*. https://doi.org/10.1145/3297858.3304059

[35] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-Aided Compiler for Low-Power Spatial Architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. https://doi.org/10.1145/2594291.2594339

[36] Yuriy Polyakov, Kurt Rohloff, and Gerard W Ryan. 2017. PALISADE lattice cryptography library user manual. *Cybersecurity Research Center, New Jersey Institute of Technology (NJIT), Tech. Rep* (2017).

[37] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proc. IEEE* (2005). https://doi.org/10.1109/JPROC.2004.840306

[38] Racket. [n.d.]. The Racket programming language. racketlang.org

[39] A. Rastogi, M. A. Hammer, and M. Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *2014 IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2014.48

[40] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent Lee, Gu-Yeon Wei, Hsien-Hsin S Lee, and David Brooks. 2020. Cheetah: Optimizations and Methods for Privacy Preserving Inference via Homomorphic Encryption. *arXiv preprint arXiv:2006.00505* (2020).

[41] Jason Reed and Benjamin C. Pierce. 2010. Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. https://doi.org/10.1145/1863543.1863568

[42] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An Architecture for Computing on Encrypted Data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. https://doi.org/10.1145/3373376.3378523

[43] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. 2010. Airavat: Security and privacy for MapReduce.. In *NSDI*, Vol. 10. 297–312.

[44] SEAL 2020. Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

[45] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. https://doi.org/10.1145/2491956.2462195

[46] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. *Combinatorial Sketching for Finite Programs*. Ph.D. Dissertation. https://doi.org/10.1145/1168857.1168907

[47] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar. 2015. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In *2015 IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2015.32

[48] Paul Syverson. 1994. A taxonomy of replay attacks [cryptographic protocols]. In *Proceedings The Computer Security Foundations Workshop VII*. IEEE, 187–191.

[49] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. https://doi.org/10.1145/2509578.2509586

[50] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit.

[51] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE.

[52] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. *IACR Cryptol. ePrint Arch.* 2015 (2015), 1153.