Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption

Meghan Cowan  
Facebook Reality Labs Research  
meghancowan@fb.com

Deeksha Dangwal  
Facebook Reality Labs Research  
ddangwal@fb.com

Armin Alaghi  
Facebook Reality Labs Research  
alaghi@fb.com

Caroline Trippel  
Stanford University  
trippel@stanford.edu

Vincent T. Lee  
Facebook Reality Labs Research  
vtlee@fb.com

Brandon Reagen  
New York University  
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 — and HE DSL named Quill to automatically generate 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. Quill captures the underlying HE operator behavior that enables Porcupine to reason about the complex trade-offs imposed by the challenges and generate optimized, verified HE kernels.

To improve synthesis time, we propose a series of optimizations including a sketch design tailored to HE and instruction restriction to narrow the program search space. We evaluate Porcupine using a set of kernels and show speedups of up to 51% (11% 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.

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 private actual content. 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 [33, 41] and custom hardware [37, 39]. The performance results are encouraging with some suggesting that HE can approach real-time latency for certain applications with sufficiently large hardware resources [37]. 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) 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, Brakerski/Fan-Vercauteren (BFV) cryptosystem [16] operates on vectors that packing multiple data 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, 11, 13, 14]. Others have proposed mechanisms to optimize data layouts for neural networks [14]. Prior solutions have also used a mix of symbolic execution [3] and rewrite rules [6, 11, 13] for code generation and optimizations for logic minimization (e.g., Boolean logic minimization [11, 26]. 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 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] so that our synthesizer completes a sketch, or HE kernel template. We introduce a novel local rotate 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 runtime (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 logic depth, the current best practice for optimizing HE programs [3, 11, 26]. 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 51% 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 51% (11% 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 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 [7–9, 16, 19] for the more technical details of how HE works.

2.1 Homomorphic Encryption Basics

Homomorphic encryption enables arbitrary computation over encrypted data or ciphertexts [18]. 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 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 [16]. 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, 16].
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. BFV supports variants of add and multiply that operate on a ciphertext and plaintext instead of two ciphertexts.

Consider two vectors of integers $X = \{x_0, x_1, \ldots, x_{n-1}\}$ and $Y = \{y_0, y_1, \ldots, y_{n-1}\}$ with ciphertext representation $X'$ and $Y'$ respectively. SIMD add and multiply both perform element-wise operations over slots. SIMD add computes $\text{add}(X', Y')$ such that $\text{decrypt}(\text{add}(X', Y'), k) = \{x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}\}$, where $k$ is the key used for encryption of $X'$ and $Y'$. Similarly, the SIMD multiply instruction processes $\text{mult}(X, Y)$ so that $\text{decrypt}(\text{mult}(X', Y'), k) = \{x_0 \times y_0, x_1 \times y_1, \ldots, x_{n-1} \times y_{n-1}\}$. Note that the underlying operations that implement $\text{add}(X', Y')$ and $\text{mult}(X', Y')$ 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 $X' = \{x_0, x_1, x_2, \ldots, x_{n-1}\}$ by one element to the left returns $\{x_1, x_2, \ldots, x_{n-1}, x_0\}$.

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 [44].

![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.](image-url)

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 [14, 25, 37].

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 \texttt{_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 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 captures the semantics of HE instructions and their latency-noise behavior, enabling Porcupine to reason about HE program behavior and noise growth. Quill is the language that input sketches are written in and used by the synthesis engine to infer HE instruction behavior as well as verify correctness. Quill currently supports BFV [16] HE, however the techniques are general and can be extended to other ring-based HE schemes, e.g., BGV [9] and CKKS [12].

Quill is used to describe straight-line HE programs that manipulate state initially defined by input vectors (either ciphertext and plaintext). Quill programs are behavioral models and not true/real 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.
**Reference Implementation**

```python
def pixelate(img, height, width):
    result = 0
    for w in 0 to width:
        for h in 0 to height:
            result[h,w] += img[h+kw, w+kw] * filter[kh, kw]
    return result
```

**Data Layout**

```
image
filter
result
```

**Kernel Specification**

```
def pixelate(img, height, width):
    result = 0
    for w in 0 to width:
        for h in 0 to height:
            result[h,w] += img[h+kw, w+kw] * filter[kh, kw]
    return result
```

**Kernel Sketch**

```
c = ??instr
mul-ct-ct (??ct) (??ct) = ??instr
add-ct-ct (??ct) = ??instr
c1 = ??instr
c2 = ??instr
```

**Quill HE DSL**

```
add-ct-ct(a,b)  a+b
sub-ct-ct(a,b)  a-b
mul-ct-ct(a,b)  axb
. . .
rot-ct(a,n)    ... = ??instr
c2 = ??instr
…
cL-1 = ??instr         
kernel
```

**Synthesized HE Kernel**

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

**Data Layout**

- **Image**: inputs and outputs must adhere to.
- **Result**: used as inputs and outputs.

**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.

**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 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 [41].

### 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 [35] that define the plaintext computation. We later use Rosette [45] 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 $3\times3$ filter.

```racket
(define (Gx img height width):
  for h in 0 to height:
    for w in 0 to weight:
      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...
Table 1. Quill instructions and their affect on the data (denoted by \textit{.data}) and noise (denoted by \textit{.noise}) of the resulting ciphertext.

| Instruction | Computation | Description | Multiplicative depth |
|-------------|-------------|-------------|----------------------|
| Add(\textit{ct}_x, \textit{ct}_y) → \textit{ct}_z | \textit{ct}_x.data + \textit{ct}_y.data | Adds two ciphertexts | max(\textit{ct}_x.n\textit{oise}, \textit{ct}_y.n\textit{oise}) |
| Add(\textit{ct}, \textit{pt}) → \textit{ct}_z | \textit{ct}.data + \textit{pt}.data | Adds a ciphertext and plaintext | \textit{ct}.noise |
| Subtract(\textit{ct}_x, \textit{ct}_y) → \textit{ct}_z | \textit{ct}_x.data - \textit{ct}_y.data | Subtract two ciphertexts | max(\textit{ct}_x.n\textit{oise} + \textit{ct}_y.n\textit{oise}) |
| Subtract(\textit{ct}, \textit{pt}) → \textit{ct}_z | \textit{ct}.data - \textit{pt}.data | Subtract a plaintext from a ciphertext | \textit{ct}.noise |
| Multiply(\textit{ct}_x, \textit{ct}_y) → \textit{ct}_z | \textit{ct}_x.data \times \textit{ct}_y.data | Multiple two ciphertexts | max(\textit{ct}_x.n\textit{oise}, \textit{ct}_y.n\textit{oise}) + 1 |
| Multiply(\textit{ct}, \textit{pt}) → \textit{ct}_z | \textit{ct}.data \times \textit{pt}.data | Multiply a ciphertext and plaintext | \textit{ct}_x.n\textit{oise} + 1 |
| Rotate(\textit{ct}, \textit{x}) → \textit{ct}_z | \textit{ct}.data[\textit{i}] \leftarrow \textit{ct}.data[(\textit{i} + \textit{x})modN] | Rotate a ciphertext \textit{x} slots to the left | \textit{ct}.noise |

4.4 Sketch

The third input a user provides is a sketch which is a template HE kernel written in Quill 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 \textit{G}_x kernel is shown below:

```sql
; Program sketch of \textit{L} components 
; \textit{ct}_0 is a ciphertext input 
(define (\textit{G}_x-Sketch \textit{ct}_0 \textit{L})
  ; choose an existing ciphertext 
  (define (??\textit{ct}) (choose* \textit{ct}))
  ; choose a rotation amount in range (0,\textit{N}) 
  (define (??\textit{r}) (apply choose* (range 0 \textit{N})))
  ; choose a rotation of an existing ciphertext 
  (define (??\textit{rot-ct})
    (rot-ct ??\textit{ct} ??\textit{r}))
  ; choose an opcode with operand holes 
  (for/list \textit{i} = 1 to \textit{L}
    (choose*
      (add-ct-ct (??\textit{ct-r} (??\textit{ct-r}))
      (sub-ct-ct (??\textit{ct-r} (??\textit{ct-r})))
      (mul-ct-ct ??\textit{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 \textit{L} components or arithmetic instructions. In this example the components are: add two ciphertexts, subtract two ciphertexts or multiply a ciphertext by 2. Each component contains holes for their instruction dependent operands. Specifically, ??\textit{ct} is ciphertext hole that can be filled with the ciphertext input or a ciphertexts generated by previous components. ??\textit{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 \textit{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. 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 \textit{L}, and 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.

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 \textit{L}) programs and iteratively explores larger programs by incrementing \textit{L} until a solution is found.
Solution. A solution is a completed sketch that matches 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:

\[
\begin{align*}
c_1 &= (\text{add-ct-ct}(\text{rot-ct } c0 -5) \ c0) \\
c_2 &= (\text{add-ct-ct}(\text{rot-ct } c1 \ 5) \ c1) \\
c_3 &= (\text{sub-ct-ct} \ (\text{rot-ct } c2 \ 1) \ (\text{rot-ct } c2 \ -1))
\end{align*}
\]

5 Synthesis Engine

This section describes how Porcupine’s synthesis engine (see Algorithm 1) searches the program space (described by our local rotate) 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 [24, 42]. It uses 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 \(\text{sketch}_L\) that encodes programs using \(L\) components. Specifically, Porcupine searches for a solution \(\text{sol}_0\) contained in \(\text{sketch}_L\) that minimizes \(L\) and satisfies the specification for all inputs.

We follow a synthesis procedure similar to those proposed in [21, 24], 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 rather then verify it is correct for all inputs; we then apply feedback to the synthesis query if verification fails.

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 \(\text{solvect}(\text{sketch}_L \ (x_0) = y_0)\), 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 specification 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 \(\text{verify}(\text{sol}(\hat{x}) = \text{spec}(\hat{x}))\).

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: \(\text{cost}(p) = \text{latency}(p) \times (1+\text{mdepth}(p))\). We include multiplicative depth to penalize high-noise programs, which can lead to larger HE parameters and lower performance.

Cost Minimization. Once a solution \(\text{sol}_0\) with cost \(\text{cost}_0\) is found, we iteratively search for a new program with lower cost (line 19). Porcupine does this by re-issuing the synthesis query with an additional constraint that ensures a new solution \(\text{sol}_1\), has lower cost: \(\text{cost}_1 < \text{cost}_0\) (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:  \( y_0 \leftarrow \text{spec}(x_0) \)  \( \triangleright \) Random input-output example
7:  \( \hat{y} = \text{spec}(\hat{x}) \)  \( \triangleright \) Symbolic input-output examples = \( \{(x_0, y_0)\} \)
8: while true do
9:  sol \leftarrow \text{solve}(\text{sketch s.t. } y = \text{sketch}(x))
10:  if sol is unsat then
11:     return False  \( \triangleright \) Sketch too restrictive
12:  if cex = \text{unsat} then
13:     return sol
14:  sol = \text{solution}(\hat{x})
15:  c = \text{cost}(\text{sketch})
16:  sol = \text{solve}(\text{sketch s.t. } y = \text{sketch}(x) \& c < c)
17:  examples.append((x, y))
18: Minimize cost
19: function OPTIMIZE
20:  sol = \text{synthesize}()
21:  c’ = \text{cost}(\text{sketch})
22:  sol’ = sol
23: while sol’ is sat do
24:  c = \text{cost}(sol), sol’ = sol’
25:  sol’ = \text{solve}(\text{sketch s.t. } y = \text{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 [41]. 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 \texttt{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 \( G_k \) kernel each output element only depends on its neighbors in the \( 3 \times 3 \) 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 \( \texttt{rot} \) to align elements covered by the window. The tree restriction reduces 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.

6.3 Multi-step synthesis

One of the limitations of program synthesis is its inability to scale to large kernels [22]. 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 under a few minutes.

7.1 Methodology

Porcupine is implemented with Rosette v3.1 [45], and configured to use Boolector [10] as its backend SMT solver. Synthesized kernels are compiled down to SEAL v3.5’s BFV library [41]. We time out all synthesis jobs after 20 minutes of no progress and return the current best solution. When running Porcupine’s kernels, security 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 [29]. Finally, because BFV cannot 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 minimize logic depth. Logic depth minimization was chosen to reflect the state-of-the-art solution that was recently proposed for optimizing HE kernels under boolean HE schemes [26]. The paper suggests that optimizing logical computation depth also minimizes noise, as fewer successive operations intuitively compounds less noise between any input-output. 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, program depth, and run time against the hand-optimized baseline. We report run

| Kernel          | Baseline | Synthesized |
|-----------------|----------|-------------|
|                 | Instr. Depth | Instr. Depth |
| Box Blur        | 6        | 4           |
| Dot Product     | 7        | 7           |
| Hamming Distance| 6        | 6           |
| L2 Distance     | 9        | 9           |
| Linear Regression| 4       | 4           |
| Polynomial Regression| 9     | 7           |
| Gx              | 12       | 6           |
| Gy              | 12       | 7           |
| Roberts Cross   | 10       | 10          |
| Sobel           | 31       | 21          |
| Harris          | 59       | 14          |

Figure 4. 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.
time speedups in Figure 4, 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. We examined the kernels and found this is because the optimal programs for depth (baseline) and instruction count (Porcupine) are the same.

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 51% 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 27% 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 4. By leveraging Porcupine synthesized kernels, our Sobel operator and Harris corner detector were 6% and 13% 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.

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 trade-offs explored by Porcupine. Figure 5 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 depth, the baseline misses the separable kernel optimization because it was not the minimum depth solution.

We observe similar results for the $G_y$ kernel and show the synthesized and baseline programs in Figure 6. 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 $G_y$ kernel weights some of the neighbor elements by two, and the baseline substitutes the multiplication with a cheaper addition (operand $c_{11}$ in Figure 6b). The synthesized $G_y$ 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 7. Porcupine’s synthesized solutions automatically also substitutes the multiplication by 2 with an addition which is performed at $c_4$ in parallel with other additions.
Additional, the optimization phase of the synthesis engine must prove it found the best solution contained in the sketch, which took longer to search over. For most of the kernels we were able to synthesize an initial solution in under 30 seconds and synthesize an optimized solution in under 2 minutes. The Roberts cross kernel required more time, taking over 2 minutes to synthesize 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 in under 2 minutes. The Roberts cross kernel required more time, taking over 2 minutes to synthesize an optimized solution under 2 minutes. The Roberts cross kernel required more time, taking over 2 minutes to synthesize an optimized solution under 2 minutes.

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 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.

Analysis of local rotate sketches. In Section 4.4 we introduced our local rotate sketches treat rotations as instruction 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 faster than the explicit rotation sketch; the explicit rotation sketch took only 3 seconds to synthesize versus 10 seconds when using a local rotate sketch. However, when we search over larger programs the explicit rotation sketch scales poorly. Synthesizing the $G_x$ kernel took over 400 seconds to find an initial solution then over 30 minutes total when using the explicit rotation sketch. On the other hand the local rotate sketches found the same solution in about 70 seconds seconds, showing that local rotate does improve synthesis scalability and search time.

7.4 Synthesis Time

Table 3 reports the time it took to synthesize each kernel averaged 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 8 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.

8 Related Work

8.1 Compilers for Homomorphic Encryption

Recent work proposes domain-specific and general compilers for HE [3, 6, 11–14]. Prior work such as CHET [14] and nGraph-HE [6] 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 [11] and Lobster [26] only target Boolean HE constructions and propose compilation...
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 3 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     | 3        | 0.87             | 2.24           | 1270         | 680        |
| 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       |

Methods that rely on multiplicative depth minimization and synthesizing rewrite rules. EVA [13] automatically inserts low-level HE instructions such as rescale and modswitch using custom rewrite rules but requires a hand-crafted HE kernel as input. Porcupine supports vectorized HE constructions, generalizes to non-DNN computation, and is more general than the rewrite rule-based prior work since program synthesis automates some of this reasoning.

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 carries out the 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 only compiler that supports the task of automatically scheduling and mapping packed vectorized HE kernels for general computation.

### 8.2 Compilers for Privacy-Preserving Computation

Compiler support has also been proposed for other privacy-preserving techniques, such as differential privacy (DP) [15] and secure multi-party computation (MPC) [20, 47] 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 [17, 30, 38]. Other programming language techniques [4] include dynamic approaches [27, 28, 40], static checking [17, 31, 38], and machine-checked proofs [5]. 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 [23, 36, 43, 46, 48] proposes and/or evaluates general-purpose compiler for MPC.

### 8.3 Synthesizing Kernels

Prior work has also used program synthesis to generated optimized kernels for other targets. For example, Spiral [34] generates optimized DSP kernels using both inductive and deductive synthesis techniques. Swizzle Inventor [32] synthesized optimized data movement for GPU kernels from a sketch that specified that computation strategy and left data movement unspecified. Because their objective was to only optimize 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.

### 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.

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] Rajeek Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. IEEE.

[3] David W Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yurii Polyakov, Kurt Kohlhoff, and Gerard Ryan. 2019. Rumparts: A programmer-friendly system for building homomorphic encryption applications. In Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography. 57–68.

[4] Gilles Barthe, Marco Gaboardi, Justin Hsu, and Benjamin Pierce. 2016. Programming language techniques for differential privacy. ACM SIGLOG News 1, 1 (2016), 34–53.

[5] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2013. Probabilistic relational reasoning for differential privacy. In Proceedings of the 9th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 97–110.

[6] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzyński. 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. 45–56.

[7] 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.

[8] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In Advances in Cryptology—Crypto 2012. Springer, 868–886.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) 6, 3 (2014), 13.

[10] Robert Brummitt and Armin Biere. 2009. Booleto: 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.

[11] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. 2015. Armadillo: a solver for bit-vectors and arrays. In 4th International Workshop on Security in Cloud Computing. Springer, 6, 3 (2014), 13.

[12] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. Foundations and Trends® in Programming Languages 4, 1-2 (2017), 1–119.

[13] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In Advances in Cryptology—Crypto 2012. Springer, 868–886.

[14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) 6, 3 (2014), 13.

[15] Robert Brummitt and Armin Biere. 2009. Booleto: 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.

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

[17] 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. 357–370.

[18] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing (Bethesda, MD, USA) (STOC ’09). ACM, New York, NY, USA, 169–178.

[19] Craig Gentry. 2010. Computing Arbitrary Functions of Encrypted Data. Commun. ACM 53, 3 (March 2010), 97–105.

[20] 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.

[21] Sumit Gulwani, Sumit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. ACM SIGPLAN Notices 46, 6 (2011), 62–73.

[22] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. Foundations and Trends® in Programming Languages 4, 1-2 (2017), 1–119.

[23] Sumit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In 2010 ACM/IEEE 32nd International Conference on Software Engineering, Vol. 1. IEEE, 215–224.

[24] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In Advances in Cryptology—Crypto 2012. Springer, 868–886.

[25] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. ACM Transactions on Computation Theory (TOCT) 6, 3 (2014), 13.

[26] Robert Brummitt and Armin Biere. 2009. Booleto: 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.

[27] Sergio 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.

[28] 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.

[29] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: an encrypted vector arithmetic library and compiler for efficient homomorphic computation. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. 546–561.

[30] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Krístin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. 2019. CHET: an optimizing compiler for fully-homomorphic neural-network inference. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. 142–156.

[31] 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.

[32] Phichtaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Chike Abuah, Tim Stevens, Pranav Gadewar, Shan, et al. 2019. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. Proceedings of the ACM on Programming Languages 3, OOPSLA (2019), 1–30.

[33] Yury Polyakov, Kurt Kohlhoff, and Gerard W Ryan. 2017. PALISADE lattice cryptography library user manual. Cybersecurity Research Center, Microsoft, 2017.
M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen

New Jersey Institute of Technology (NJIT), Tech. Rep (2017).

[34] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[35] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[36] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[37] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[38] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[39] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[40] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[41] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[42] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[43] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[44] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[45] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[46] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[47] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.

[48] Markus Puschel, José MF Moura, Jeremy R Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, et al. 2005. SPIRAL: Code generation for DSP transforms. Proc. IEEE 93, 2 (2005), 232–275.