BitPacker: Enabling High Arithmetic Efficiency in Fully Homomorphic Encryption Accelerators

Nikola Samardzic
nsamar@csail.mit.edu
Massachusetts Institute of Technology
Cambridge, MA, USA

Daniel Sanchez
sanchez@csail.mit.edu
Massachusetts Institute of Technology
Cambridge, MA, USA

Abstract

Fully Homomorphic Encryption (FHE) enables computing directly on encrypted data. Though FHE is slow on a CPU, recent hardware accelerators compensate most of FHE's overheads, enabling real-time performance in complex programs like deep neural networks. However, the state-of-the-art FHE scheme, CKKS, is inefficient on accelerators. CKKS represents encrypted data using integers of widely different sizes (typically 30 to 60 bits). This leaves many bits unused in registers and arithmetic datapaths. This overhead is minor in CPUs, but accelerators are dominated by multiplications, so poor utilization causes large area and energy overheads.

We present BitPacker, a new implementation of CKKS that keeps encrypted data packed in fixed-size words, enabling near-full arithmetic efficiency in accelerators. BitPacker is the first redesign of an FHE scheme that targets accelerators. On a state-of-the-art accelerator, BitPacker improves performance by gmean 59% and by up to 3×, and reduces energy by gmean 59%. BitPacker does not reduce precision and can be applied to all prior accelerators without hardware changes.

CCS Concepts: **• Security and privacy → Cryptography; • Computer systems organization → Architectures.**

ACM Reference Format:
Nikola Samardzic and Daniel Sanchez. 2024. BitPacker: Enabling High Arithmetic Efficiency in Fully Homomorphic Encryption Accelerators. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3620665.3640397

1 Introduction

Fully Homomorphic Encryption (FHE) enables computing directly on encrypted data. FHE allows offloading computation to third parties, like untrusted servers, with guaranteed privacy. Recent work has applied FHE to many compelling domains, including deep learning inference, genome analysis, and private information retrieval [8, 16, 18, 21, 22, 24, 28].

The key challenge limiting the adoption of FHE is its high overhead: on a CPU, FHE programs are about 10,000× slower than their unencrypted counterparts. Fortunately, FHE is very amenable to hardware acceleration. Recent FHE accelerators [27, 29, 39, 40] achieve speedups of well over 1,000×, closing most of FHE's performance gap and enabling new applications, such as real-time private deep learning.

While FHE accelerators are enticing, they leave significant performance on the table because they accelerate FHE algorithms that were designed and optimized for CPUs. FHE accelerators have different tradeoffs than CPUs, making *hardware-software codesign* necessary: higher efficiency is possible by *redesigning algorithms to use accelerators well.*

In this paper, we present the first significant redesign of an FHE scheme that targets accelerators. We focus on CKKS, the state-of-the-art FHE scheme. CKKS supports arithmetic on encrypted vectors of fixed-point values. CKKS is ideally suited to machine learning, and is the scheme used by most accelerators and applications [8, 16, 18, 21, 22, 24, 28].

CKKS encodes encrypted data using very large integers, typically over 1,000 bits wide. Efficient CKKS implementations use Residue Number System (RNS) encoding [15] to represent each wide integer using multiple narrow integers,

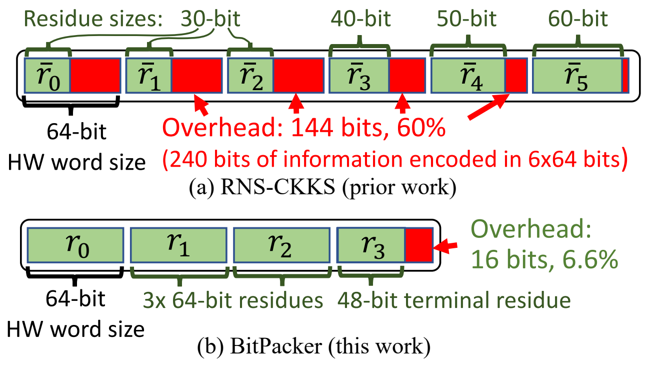

Figure 1. The state-of-the-art RNS-CKKS implementation leaves many bits of the datapath unused, wasting area and energy, because it links residue size and scale (a). BitPacker is a new RNS implementation of CKKS that avoids this problem, achieving high utilization (b).

called *residues*. RNS makes operations more efficient, and enables each narrow residue to fit within a hardware word.

However, CKKS implementations *leave many bits of the datapath unused*: typical CKKS programs use 30- to 60-bit residues. This incurs large overheads for FHE accelerators, as they are dominated by (modular) multiplications, whose area and energy grow *quadratically* with bitwidth.

This inefficiency arises because RNS-CKKS [9], the current state-of-the-art implementation, constrains the size of the residues. Specifically, residues must equal the *scale*, a CKKS parameter that determines the precision (number of usable bits) of encrypted data. Applications typically use scales between 30 and 60 bits, and since larger scales are expensive (Sec. 2), *each application combines a wide range of scales.*

Fig. 1(a) shows this problem with a simple example. It shows how RNS-CKKS encodes a ciphertext with 6 scales, which range from 30 to 60 bits, when using a 64-bit datapath. RNS-CKKS maps each scale to one residue, taking 6 64-bit words per wide integer. This is a 60% space overhead, since the integer encodes 240 ($3 \cdot 30+40+50+60$) bits of information.

FHE accelerators suffer heavy performance and energy penalties from unused datapath bits: for example, a 60% space overhead causes 2.6× energy overheads in multipliers. Prior work has thus carefully tuned datapath width. BTS and ARK [27, 29] use a 64-bit datapath, as in Fig. 1(a). F1 and CraterLake [39, 40], use narrower datapaths (~ 32 bits), which are more efficient, and represent each scale that exceeds the datapath width using two smaller residues (e.g., encoding a 50-bit scale with two 25-bit residues), a technique called double-prime rescaling. And SHARP [26] carefully chooses a 36-bit datapath to make double-prime rescaling less frequent.

However, *RNS-CKKS incurs high overheads even with carefully tuned datapath widths*, because it's impossible for a fixed datapath to be efficient across residue sizes. For example, consider Fig. 1(a): a 32-bit datapath would be efficient for 30-bit scales, but very inefficient for 40-bit scales.

We present BitPacker, a new RNS implementation of CKKS that *decouples scale and residue size*. BitPacker packs ciphertexts into residues sized to the datapath width, using hardware efficiently, while supporting *any* scale. Fig. 1(b) shows this representation and how it reduces overheads.

BitPacker relies on several novel contributions. First, we contribute novel implementations of key CKKS procedures (rescaling and mod-switching) that enable fully packing residues while retaining precision (Sec. 3.2). Second, we contribute a novel algorithm that selects tightly packed residue moduli (Sec. 3.3). Third, we design an accelerator-friendly implementation that reuses the functional units of state-of-the-art designs like CraterLake, ARK, and SHARP (Sec. 4), and requires no hardware changes.

BitPacker's benefits are especially relevant to FHE accelerators (Sec. 4). By using fewer hardware words to represent (and compute on) the same data, BitPacker improves performance and energy superlinearly. BitPacker also reduces on-chip storage and functional unit area, as ciphertexts are smaller. While BitPacker makes all datapath widths more efficient, it makes narrow (28-bit) datapaths the most efficient choice, simplifying accelerator design.

We evaluate BitPacker on FHE accelerators with datapaths ranging from 28 to 64 bits (Sec. 6). BitPacker improves performance by 59% on average over the 28-bit design, and by up to 3×, on a diverse set of benchmarks and across word sizes. BitPacker also reduces energy by 59% and data movement by 37%. BitPacker improves performance on CPUs as well, by 24% on average. Finally, BitPacker does not hurt accuracy.

2 Background and Motivation

In this section, we first present the necessary background on FHE. Then, we present the CKKS scheme and RNS-CKKS [9], its RNS implementation. RNS-CKKS's inefficient use of fixed-width words motivates the need for BitPacker.

2.1 CKKS is the state-of-the-art FHE scheme

There are several FHE *schemes* that encrypt data of different types. There are two broad types of schemes: *vector* schemes (e.g., BGV [6], B/FV [14], and CKKS [10]) encrypt a large vector of numbers in each ciphertext; and *scalar* schemes (e.g., FHEW/TFHE [11, 12]) encrypt a single value per ciphertext. Scalar schemes are more flexible than vector ones, but they have much higher overheads. Thus, for applications that use vectors well, like machine learning, vector schemes are preferable.

Vector schemes provide three operations on encrypted vectors: elementwise addition, elementwise multiplication, and rotations. Vector schemes differ in their supported data type: in BGV and B/FV, plaintext vector elements are integers modulo a certain value. In CKKS, vector elements are fixed-point values.

For most applications, CKKS's support of fixed-point arithmetic is a much better fit than the modular integers of earlier schemes. Thus, CKKS is the most widely used scheme, with applications in deep learning inference, machine learning, genomics, and other areas [8, 16, 18, 21, 22, 24, 28]. State-of-the-art accelerators (Sec. 4.1) target CKKS.

2.2 CKKS implementation

We now present the key implementation characteristics of CKKS. Full details are available elsewhere [10, 36, 40].

Datatypes: Fig. 2 shows CKKS's plaintext and ciphertext datatypes, and its encryption process. Each plaintext is a vector of n fixed-point numbers. The *scale* parameter, S, determines the width of the mantissa (the fractional part) of each element.

Each ciphertext is a pair of polynomials (`ct.0` and `ct.1`). Each polynomial has $N = 2n$ integer coefficients modulo a large value Q. Each ciphertext is very large for two reasons.

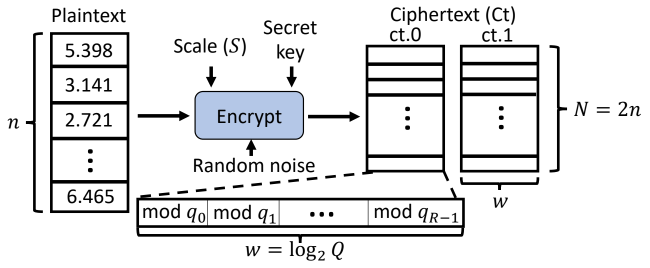

Figure 2. Encryption process for CKKS.

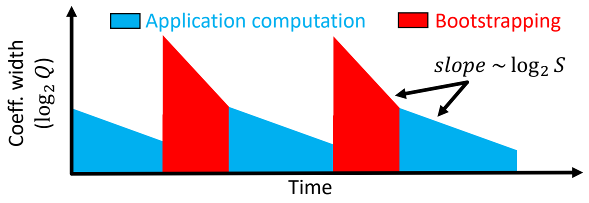

Figure 3. Noise management affects ciphertext coefficient width ($\log_2 Q$). Narrowing $\log_2 Q$ after rescales trims noise; bootstrapping produces a low-noise, high-$\log_2 Q$ ciphertext.

First, N must be large for security ($N = 65,536 = 2^{16}$ is typical) [33]. Second, each coefficient is very wide, typically over 1,000 bits, much wider than the scale S.

Noise and precision: To prevent decryption, each ciphertext is encrypted with some noise, as Fig. 2 shows. In CKKS, this noise introduces a small amount of error into the encrypted data. This affects precision: for a given scale S, which has $\log_2 S$ bits, the mantissa has between $\log_2 S - 20$ and $\log_2 S - 15$ usable (error-free) bits, with the lower bits taken over by noise. In practice, programs typically use scales ranging from 30 to 60 bits (i.e., 10 to 45 error-free mantissa bits).

Homomorphic operations: Ciphertexts support only three operations, called *homomorphic operations*: elementwise *adds*, elementwise *multiplies*, and *rotates* of the underlying encrypted vectors (add and multiply allow one of their operands to be unencrypted). Homomorphic operations have different costs: addition is cheap, but multiplication and rotation are expensive; we discuss their costs in Sec. 4.

Noise management: Ciphertexts are encrypted with a small amount of noise. Unfortunately, homomorphic operations increase noise, and if noise becomes too large, it corrupts the encrypted values. Noise grows primarily with multiplies. Specifically, multiplying two ciphertexts with scale S and noise δ produces a result with scale S^2 and noise $\sim S\delta$.

CKKS uses three techniques to manage noise and scale: rescaling, adjusting, and bootstrapping (these operations do not change the underlying encrypted values):

1. Rescaling: A rescale divides the coefficients of the ciphertext by a value d, which can be any divisor of modulus Q. This reduces Q, the noise, and the scale by a factor d. Thus, by rescaling by a value $d \approx S$ after a multiplication, we can reset the scale back to $S^2/d \approx S$ and the noise back to $\delta S/d \approx \delta$.

2. Adjusting: Since rescaling changes Q, CKKS uses a procedure called *adjust* that trims down Q by a factor d but does not change the scale by a factor d. Adjusting enables performing operations between ciphertexts that have undergone a different number of multiplications (and rescales).

The following example illustrates the use of rescale and adjust. Suppose we want to compute $x^2 + x$, where x is a ciphertext with a 1000-bit Q and a 50-bit scale ($S = 2^{50}$). Multiplying $x * x$ grows the scale to $S^2 = 2^{100}$, or 100 bits; by choosing a divisor of Q, d close to 2^{50}, $\texttt{rescale}(x * x, d)$ trims 50 bits of the modulus and the scale, leaving a 950-bit modulus $Q/d \approx 2^{950}$ and a 50-bit scale $S^2/d \approx 2^{50}$. But we cannot add x directly to this ciphertext, because their moduli and scales are different ($Q \neq Q/d$ and $S \neq S^2/d$).

$\texttt{adjust}(x, d)$ produces a compatible ciphertext with a modulus Q/d and scale S^2/d. The final result, $\texttt{rescale}(x * x, d) + \texttt{adjust}(x, d)$, has modulus Q/d and scale S^2/d.

3. Bootstrapping: Once coefficients cannot be narrowed further, the ciphertext must be *bootstrapped*, a procedure that restores the ciphertext to a large Q, letting it undergo more operations. Bootstraps let FHE support arbitrarily large programs, but are very expensive, involving hundreds of rotates and multiplies. Thus, they often dominate performance.

Fig. 3 shows how Q's bitwidth ($\log_2 Q$) evolves over a typical FHE application: Q progressively decreases as rescales trim noise, and is reset back up by bootstrapping.

Leveled execution: Since multiplications consume Q in chunks of $\sim \log_2 S$ bits, it's common to think of noise budget in terms of discrete *levels*. For example, if each multiplication consumes 50 bits of Q, a 1000-bit Q supports a multiplicative depth of $L = \log_2 Q / \log_2 S = 1000/50 = 20$ levels.

Using multiple scales: Choosing the smallest possible scale that does not hurt precision is critical to performance. A large scale consumes Q more rapidly, so for a program of a fixed multiplicative depth, a larger scale requires using larger Q and eventually causes more frequent bootstrapping.

We have so far discussed using a single scale throughout the program. However, application developers tune the scale to balance precision and cost, and often *use different scales at different levels*. Thus, the same application uses a wide range of scales. Bootstrapping causes much of this variability: several of its operations need high precision, and use 55- to 60-bit scales. By contrast, non-bootstrap operations often use lower scales, 30 to 45 bits. Fig. 3 shows this: by using lower precision in non-bootstrap computation, a program consumes Q more slowly and makes bootstrapping less frequent. Thus, *it is crucial to support a wide range of scales efficiently.*

2.3 RNS implementation of CKKS

Residue Number System (RNS) representation: All high-performance implementations of CKKS use RNS representation [15]. RNS allows encoding each ciphertext polynomial with wide coefficients as R *residue polynomials* with narrow

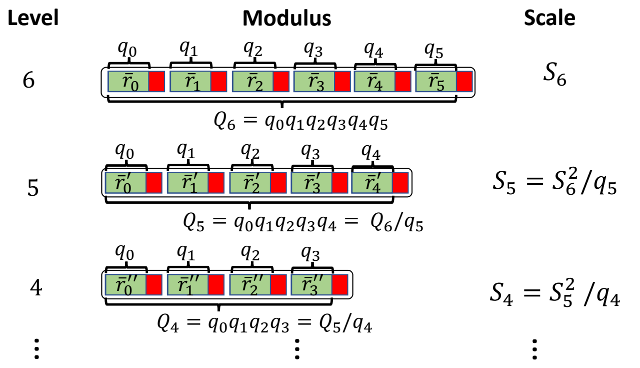
Level
Modulus
Scale
6
q_0 q_1 q_2 q_3 q_4 q_5
$\bar{r}_0$ $\bar{r}_1$ $\bar{r}_2$ $\bar{r}_3$ $\bar{r}_4$ $\bar{r}_5$
$Q_6 = q_0q_1q_2q_3q_4q_5$
S_6
5
$\bar{r}'_0$ $\bar{r}'_1$ $\bar{r}'_2$ $\bar{r}'_3$ $\bar{r}'_4$
$Q_5 = q_0q_1q_2q_3q_4 = Q_6/q_5$
$S_5 = S_6^2/q_5$
4
$\bar{r}''_0$ $\bar{r}''_1$ $\bar{r}''_2$ $\bar{r}''_3$
$Q_4 = q_0q_1q_2q_3 = Q_5/q_4$
$S_4 = S_5^2/q_4$

Figure 4. Evolution of residue moduli and scales across three adjacent levels for RNS-CKKS.

```
def rnsCkksRescale(ct: Ct) -> Ct:
  for x in (ct.0, ct.1):
    # x has shape [L][N]
    # all ops are on N-element vectors
    for i in [0:L-2]:
      F = 1 / q[i-1] mod q[i] # precomputed
      x[i] = (x[i] - x[L-1]) * F mod q[i]
    return ct[0:L-2] # drop last residue
```

Listing 1. Implementation of RNS-CKKS rescale from L to $L-1$.

```
def rnsCkksAdjust(ct: Ct) -> Ct:
  # ct's scale is S[L] == scale at level L
  K = q[L-1] * S[L-1] / S[L] # precomputed
  # ct0's scale = S[L] * K = q[L-1] * S[L-1]
  ct0 = mulConst(ct, K)
  # scale after rescale = S[L-1]
  return rnsCkksRescale(ct0)
```

Listing 2. Implementation of adjust in RNS-CKKS between levels L and $L-1$.

coefficients. This is achieved by choosing the wide modulus Q to be a product of R smaller factors, $Q = q_0q_1...q_{R-1}$, called *residue moduli.* Then, each coefficient x mod Q is represented as a collection of *residues* (x mod q_0, ..., x mod q_{R-1}). We denote these residues as $r_i = x$ mod q_i (Fig. 2).

RNS representation reduces overall operation cost, and allows supporting many coefficient widths with a single narrow width in hardware. Existing software libraries [1, 3, 19] and accelerators [26, 27, 29, 37, 39, 40] all use RNS.

RNS-CKKS links scales and residues: As originally proposed, RNS-CKKS [9] makes R=L, linking each level and scale to a particular residue modulus. Ciphertexts at level L have modulus $Q_L = q_0...q_{L-1}$, and their scale is $\sim q_L$. For instance, to implement a 1000-bit Q with 50-bit scale, we would choose $L = 20$ residue moduli $q_0, ..., q_{19}$ close to 2^{50}.

Fig. 4 shows how residues and scales evolve in RNS-CKKS. In this example, the ciphertext starts at level $L = 6$ with scale S_6. Each ciphertext coefficient consists of 6 residues, with moduli $q_0...q_5$. After a multiplication and rescale, the resulting $L = 5$ ciphertext has scale $S_5 = S_6^2/q_5$, and each coefficient consists of 5 residues, with moduli $q_0...q_4$. In other words, rescaling is achieved by reducing the number of residues.

RNS-CKKS links residues to scales to make rescaling and

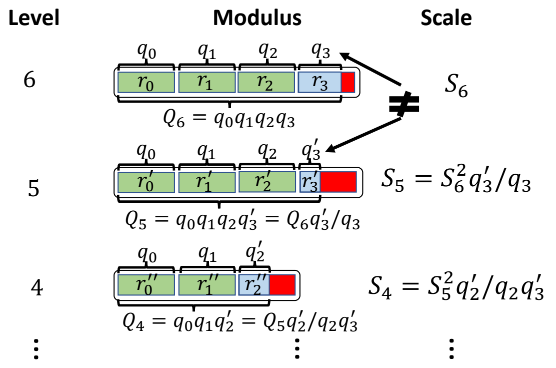

Figure 5. Evolution of moduli and scales across three adjacent levels for BitPacker.

adjusting implementable, as we describe below. But this representation *limits hardware efficiency*, because it leaves much of the datapath unused, as Fig. 1 showed.

Rescaling and adjusting: RNS-CKKS representation makes rescaling cheap. Listing 1 shows the pseudocode: for each coefficient, the last residue is blended with the remaining residues, then dropped. Rescaling requires $O(RN)$ multiplications, asymptotically fewer than those required by a homomorphic multiplication ($O(R^2N)$, Sec. 4).

Adjusting is similarly cheap in RNS-CKKS. The original implementation [9] proposed a trivial but approximate adjust procedure, called *mod-down*: adjusting to level *dst* from *src* is done by discarding residues $r_{dst}, ..., r_{src-1}$. This discarding does not change the scale or the underlying encrypted values. However, this introduces error because the scales at different levels are not quite the same, as Fig. 4 shows: $S_5 = S_6^2/q_5 \neq S_6$. This error is negligible with very large residue moduli (e.g., $\sim 2^{50}$), but not with smaller ones (e.g., $\sim 2^{30}$).

Listing 2 shows Kim et al.'s adjust implementation[25], which avoids mod-down's error: it adjusts the scale so that the result has the same scale as that of rescale. Specifically, an adjust of a ciphertext at level L and scale S_L produces a ciphertext at level $L-1$ with scale S_{L-1} equal to the result of rescaling a ciphertext at level L with scale S_L^2 (i.e., $S_{L-1} = S_L^2/q_{L-1}$; Fig. 4). This ensures all ciphertexts at the same level also have the same scale, and can thus be safely added.

`rnsCkksAdjust`'s implementation is almost identical to `rnsCkksRescale`, except that the scale is adjusted by multiplying all coefficients with a constant. To support adjusting between two non-adjacent levels, Kim et al. discard residues until the level is one higher than the target level; then Listing 2 is applied. Our evaluation uses this implementation.

Supporting scales beyond the hardware word size: As described so far, RNS-CKKS is limited to scales that fit within a hardware word. A simple extension, multiple-prime rescaling [26, 40] avoids this limitation by using multiple residue moduli per scale. For example, a 64-bit datapath can support a 72-bit scale by using two 36-bit residues. This allows using narrow datapaths, but still suffers poor datapath utilization.

3 BitPacker Representation

3.1 Overview

BitPacker is an implementation of CKKS that leverages RNS representation more effectively than RNS-CKKS by keeping most data packed in narrow words of a fixed size.

BitPacker decouples residues and scales: Fig. 5 shows an example of BitPacker's representation, showing how residues and scales evolve across levels. This example mimics the RNS-CKKS example in Fig. 4 to facilitate a side-by-side comparison: in both cases, the ciphertext has a 240-bit Q at $L = 6$, uses a 40-bit scale at each level, and hardware has 64-bit words. Whereas RNS-CKKS represents this ciphertext using 6 40-bit residues, one per level (Fig. 4), BitPacker represents this ciphertext using only 4 residues: the first 3 residues use moduli q_0, q_1, and q_2 close to the hardware word size, 64 bits, and the fourth residue uses modulus q_3, which has 48 bits. Using 4 residues instead of 6 yields big savings, because the cost of homomorphic operations grows superlinearly with residues (Sec. 4).

It is important to emphasize that BitPacker is a more compact representation of the *same amount of information*. Although Fig. 4 has 6 residues and Fig. 5 has 4, they both encode a 240-bit Q with 6 levels. Thus, BitPacker does not reduce the number oflevels or cause more frequent bootstrapping.

Terminal and non-terminal residues: In general, a BitPacker ciphertext consists of *(1)* several word-sized residues (like q_0, q_1, and q_2 above), called **non-terminal residues**, shown in green in Fig. 5; and *(2)* one or a small number of residues smaller than the word size (like q_3 above), called **terminal residues**, shown in blue in Fig. 5. Terminal residues let BitPacker represent coefficients with an arbitrary number of bits.

All examples in Fig. 4 show a single terminal residue, but as we will see, using more than one terminal modulus is needed sometimes (typically, no more than two are needed).

Rescale and adjust: BitPacker's representation requires new techniques for rescaling and adjusting, and our key contribution is to show that these modulus changes are possible and simple. The lack of these mechanisms is what led RNS-CKKS to link residues and scales [9].

For example, in Fig. 5, rescaling the $L = 6$ ciphertext to produce an $L = 5$ ciphertext results in coefficients that still use 4 residues: the 3 non-terminal residues use the same moduli (q_0, q_1, and q_2), but the terminal residue uses a *smaller* modulus, q'_3 ($\neq q_3$). Similarly, $L = 4$ uses three residues, with non-terminal moduli q_0 and q_1, and terminal modulus q'_2.

This example shows that, whereas RNS-CKKS only sheds residue moduli, BitPacker *introduces new ones* as it moves across levels. Our key insight is that this can be done accurately by temporarily scaling *up* the ciphertext to use a larger Q and number of residues, then shedding the unneeded ones.

BitPacker's rescale and adjust have small costs (4-7% in our evaluation), and yield large efficiency gains in return.

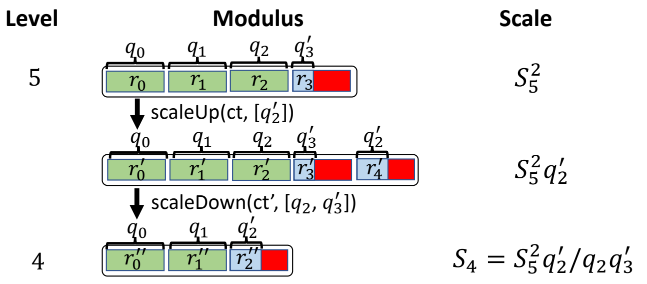

Figure 6. Example showing how BitPacker rescales the ciphertext from Fig. 5 from level $L = 5$ to level $L - 1 = 4$.

In the rest of this section, we first explain how BitPacker's rescale and adjust operations work (Sec. 3.2). Because BitPacker is a simple change in RNS representation, *all other operations are exactly the same as in RNS-CKKS.* Then, we describe how BitPacker chooses terminal and non-terminal moduli to achieve high efficiency and accuracy (Sec. 3.3). Finally, we discuss BitPacker's security, showing that it achieves the same security level as CKKS and RNS-CKKS (Sec. 3.4).

3.2 Level Management

Rescale and adjust move a ciphertext from a higher source level L_{src} to a lower destination level L_{dst}. Unlike in RNS-CKKS, in BitPacker, the ciphertext at L_{dst} does not use a subset of the residue moduli at L_{src}: its terminal moduli are different.

We now present `bpRescale` and `bpAdjust`, the BitPacker versions of rescale and adjust. These procedures use two low-level RNS operations: *scale-up* and *scale-down.*

Rescale and adjust follow the same process to change terminal moduli. Fig. 6 shows this for rescale, when moving from $L_{src} = 5$ to $L_{dst} = 4$. (Because rescaling happens after a multiplication, the initial scale of the ciphertext is S_5^2.) First, the ciphertext is *scaled up* to use a larger modulus, introducing the terminal moduli of L_{dst}. Then, the ciphertext is *scaled down* to shed the moduli at L_{src} that are not present in L_{dst}. This drops L_{src}'s terminal moduli, and may shed some non-terminal moduli too. In Fig. 6's example, rescaling first adds a new residue with terminal modulus q'_2, then sheds q'_3 and q_2.

Scale-up: To implement the first step above, we leverage a procedure called *scale-up.* A scale-up of a ciphertext with modulus Q, scale S, and noise δ by a factor d coprime with Q produces a ciphertext that encrypts the same data as the input with modulus Qd, scale Sd, and noise δd.

Listing 3 shows the implementation of the `scaleUp` procedure for an RNS representation. `scaleUp` is not our contribution: RNS-CKKS uses `scaleUp` prior to bootstrapping, to produce a much larger ciphertext that bootstrapping can denoise, as Fig. 3 showed. Our insight is that scale-ups can also be used to implement rescale and adjust.

```
# qs is a list of residue moduli to scale up by
# qs are all coprime to ct's modulus
def scaleUp(ct: Ct, qs: Seq[int]) -> Ct:
  K = product(qs) # precomputed
  ct = mulConst(ct, K)
  for x in (ct.0, ct.1):
    # Before append, x's shape is [R][N]
    x.append([[0] * N] * len(qs))
  return ct[0:R+len(qs)]
```

Listing 3. Implementation of `scaleUp`.

```
1 def bpRescale(ct: Ct) -> Ct:
2   # moduli[L] holds all residue moduli at level L
3   qs_only_in_Ldst = [qi for qi in moduli[L-1]
4                      if qi not in moduli[L]]
5   ct0 = scaleUp(ct, qs_only_in_Ldst)
6   qs_only_in_Lsrc = [qi for qi in moduli[L]
7                      if qi not in moduli[L-1]]
8   return scaleDown(ct0, qs_only_in_Lsrc)
```

Listing 4. Implementation of `bpRescale`.

```
 1 # rescaleUs = [p_0, p_1, ..., p_k]
 2 def scaleDown(ct: Ct, rescaleUs: Seq[int]) -> Ct:
 3   for x in (ct.0, ct.1):
 4     # x's shape is [R][N]
 5     x = moveResiduesToEnd(x, rescaleUs)
 6     k = len(rescaleUs)
 7     subMe = zeros(shape=(R, N))
 8     subMe[R-k:R] = x[R-k:R]
 9     for i in range(R-k, R):
10       for j in range(R-k):
11         # C[i][j] is precomputed
12         subMe[j] += C[i][j] * subMe[i] mod q[j]
13     x -= subMe
14     # InvP is precomputed
15     x = mulConst(x[0:R-k], InvP)
16   return ct[0:R-k]
```

Listing 5. Implementation of `scaleDown`.

Rescale: Listing 4 shows the implementation of BitPacker's rescale. `bpRescale` reduces the ciphertext's level by one, from L to $L-1$. First, `bpRescale` scales-up the ciphertext by all the residue moduli at $L-1$ that do not appear at L (lines 3–5). Then, `bpRescale` scales-down this intermediate ciphertext by all the residue moduli that appear at L but not at $L-1$ (lines 6–8), shedding these moduli and producing the final result.

`bpRescale` implements the scale-down by calling our `scaleDown` procedure, shown in Listing 5. `scaleDown` rescales each ciphertext polynomial x by multiple residue moduli $p_0, p_1, ..., p_{k-1}$, reducing the scale (and noise) by a factor $P = p_0 \cdot p_1 \cdot ... \cdot p_{k-1}$ and shedding these moduli. This is done by computing the floor of $x/p_0p_1 \cdot ... \cdot p_{k-1}$. Listing 5 shows how this is done efficiently in RNS. To simplify the code, line 5 reorders x's residues, placing those to be shed at the end.

`scaleDown` uses two types of precomputed values. First, `C[i][j]` are the product of q_i and the Bézout coefficient corresponding to q_i for $(q_i, P/q_i) \bmod q_j$ [5]; thus, each `C[i][j]` fits into one hardware word. Second, `InvP` is the multiplicative inverse of $P \bmod Q/P$, where Q is the product of the R residue moduli of the input `ct` [5]; thus, `InvP` takes $R-k$ words.

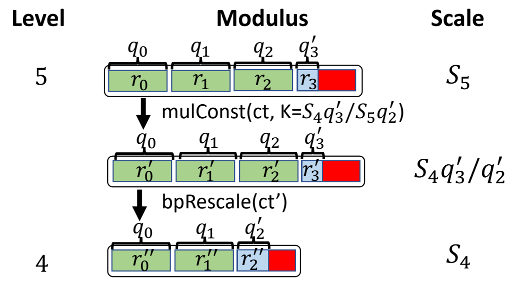

Figure 7. Example showing how BitPacker adjusts the ciphertext from Fig. 5 from level $L = 5$ to level $L - 1 = 4$.

```
1 def bpAdjust(ct: Ct) -> Ct:
2   # Q[L] is product of residue moduli at level L
3   K = Q[L] * S[L-1] / (Q[L-1]*S[L]) # precomputed
4   ct = mulConst(ct, K)
5   return bpRescale(ct)
```

Listing 6. Implementation of `bpAdjust`.

`scaleDown` is similar to calling `rnsCkksRescale` (Listing 1) multiple times, since `rnsCkksRescale` scales down one residue at a time. But shedding multiple moduli at once better leverages accelerators, as we will see in Sec. 4.

Adjust: Listing 6 shows the implementation of BitPacker's adjust, `bpAdjust`. Fig. 7 shows `bpAdjust`'s behavior, illustrating how residues and scales evolve over time (similarly to how Fig. 6 illustrates `bpRescale`).

`bpAdjust` works similarly to `rnsCkksAdjust`: it performs a scale adjustment (lines 3–4) and a rescale (line 5), except: *(1)* the scale is adjusted by $K = (Q_L/Q_{L-1}) * (S_{L-1}/S_L)$ instead of $K = q_{L-1} * (S_{L-1}/S_L)$; this is because BitPacker moduli, and thus scales, change differently across levels than in RNS-CKKS (compare scales in Fig. 5 and Fig. 4); and *(2)* we use `bpRescale` instead of `rnsCkksRescale`, again, to match moduli and scales with those produced by `bpRescale`.

BitPacker must also support adjusting down by multiple levels. This follows the same approach as RNS-CKKS: we first drop residue moduli as long as the modulus is larger than $L_{dst} + 1$'s modulus; then apply Listing 6's algorithm.

3.3 Choosing Residue Moduli

Choosing residue moduli in BitPacker is more involved than in RNS-CKKS. Fig. 8 shows the requirements that BitPacker must meet when choosing moduli. Each FHE program requires a number of *levels*, and each level has a *target scale*. At level 0, ciphertexts must have a particular modulus width $\log_2 Q_{min}$ to enable bootstrapping or decryption. The target security level (e.g. 128 bits) sets the maximum modulus width, $\log_2 Q_{max}$, and the size of each ciphertext polynomial, N (Sec. 3.4). Finally, hardware fixes the word size, w.

These constraints limit the moduli that BitPacker can use. First, residue moduli must be primes $\leq w$ bits wide, so that each residue fits in a single hardware word. Second, moduli

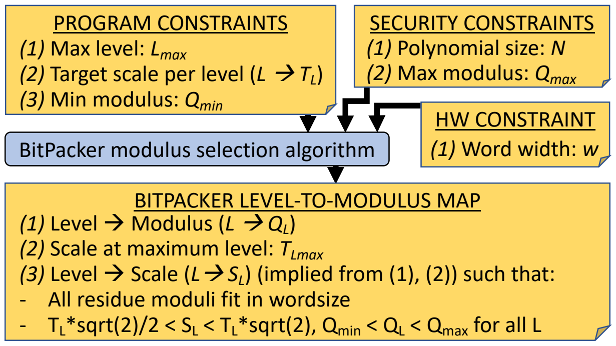
PROGRAM CONSTRAINTS
(1) Max level: L_{max}
(2) Target scale per level ($L \rightarrow T_L$)
(3) Min modulus: Q_{min}
SECURITY CONSTRAINTS
(1) Polynomial size: N
(2) Max modulus: Q_{max}
HW CONSTRAINT
(1) Word width: w
BitPacker modulus selection algorithm
BITPACKER LEVEL-TO-MODULUS MAP
(1) Level → Modulus ($L \rightarrow Q_L$)
(2) Scale at maximum level: T_{Lmax}
(3) Level → Scale ($L \rightarrow S_L$) (implied from (1), (2)) such that:
- All residue moduli fit in wordsize
- T_L*sqrt(2)/2 < S_L < T_L*sqrt(2), Q_{min} < Q_L < Q_{max} for all L

Figure 8. The modulus selection algorithm needs to take into account program, security, and hardware constraints.

must be *NTT-friendly primes* [33], i.e., prime numbers that give remainder 1 when divided by $2N$.

Unfortunately, few primes are NTT-friendly. For instance, with $N = 64K$ and $w = 28$ bits, there are only 244 NTT-friendly primes. Moreover, all NTT-friendly primes are larger than $2N$, which places a harsh lower bound on moduli size. For example, with $N = 64K$, all NTT-friendly primes are 17 bits or wider.

These constraints make it necessary to sometimes use multiple terminal moduli. For example, with $w = 28$ bits, a 70-bit coefficient needs three residues. But if we were to use two 28-bit non-terminal residues, the terminal residue would need a 14-bit modulus, which does not exist. Instead, this can be achieved with, for example, one 28-bit non-terminal residue and two 24-bit terminal residues, for which valid moduli exist.

Choosing target moduli: To generate a modulus for level L, we first set its target modulus. The target modulus at level L can be derived from the actual scale (S_{L+1}) and modulus (Q_{L+1}) at level $L + 1$, as well as the target scale at level L [25]. Because of this dependence, we define the mapping between levels and moduli starting from the top level and going down.

Listing 7 shows our algorithm to select residue moduli for a particular level. We detail its operation below.

Non-terminal moduli: Our modulus selection algorithm first chooses non-terminal moduli that keep the ciphertext as packed as possible. This is simple: with a w-bit word size, the algorithm chooses the NTT-friendly primes closest to 2^w and smaller than 2^w. The algorithm picks enough non-terminal moduli $q_0, ..., q_k$ to cover the largest ciphertext needed (i.e., k is such that $q_0 \cdot ... \cdot q_k > Q_{max}$). The algorithm also picks $q_0 > ... > q_k$, so the moduli used by more levels are larger.

Terminal moduli: Next, the modulus selection algorithm picks the terminal moduli associated with each level. It tries to minimize the distance between the level's target scale (requested by the program) and the actual scale. We use a simple greedy search, shown in Listing 7. The algorithm computes the target Q_L for each level, and tries to match it using the smallest number of terminal primes. For instance,

```python
def Greedy(target_q: Rational, result = []: Seq[int],
           primes_left = AllDescPrimes(): Seq[int])
             -> Optional[Seq[int]]:
  if target_q < sqrt(2)/2:  # Overshoot target,
    return None             # so stop (and backtrack)

  if sqrt(2)/2 < target_q < sqrt(2):
    # Product of result within 0.5 bits
    # of target_q, so return success
    return result

  # Iterate through all primes smaller than
  # those in result in decreasing order
  for idx, prime in primes_left:
    # Try greedily adding the next prime to result
    result = Greedy(target_q/prime, result + [prime],
                    primes_left[idx:])
    if result is not None:
      return result  # Stop on first success
  return None        # No match found
```

Listing 7. Greedy DFS algorithm for generating residue moduli whose product matches that of a target modulus.

for the 70-bit target in the above example, this algorithm will first try to use a single terminal prime, find there's no single 14-bit prime, and then explore combinations of two terminal primes.

Using a larger number of terminal moduli may enable finding a more precise match to the target scale, but it also makes level management more expensive. Thus, we accept the closest solution that is within 0.5 bits of the target scale (i.e., $|\log_2 S_L - \log_2 T_L| < 0.5$). We find that this works well in practice and does not impact accuracy.

Performance: This algorithm completes in less than a second for all word sizes we evaluate (Sec. 6). We use precomputed NTT-friendly primes; for $w \leq 36$ bits, we exhaustively enumerate all such primes; for larger w, we enumerate enough primes near 2^w to always use the best non-terminal primes, and sample 500 primes evenly spaced out logarithmically, to be used as candidates for terminal primes.

3.4 Security

BitPacker does not change the security of CKKS compared to RNS-CKKS, or to the original (non-RNS) CKKS implementation [10]. CKKS depends on the security of the ring learning with errors (R-LWE) problem [7]. The security of a CKKS ciphertext is proportional to $N/\log_2 Q_{max}$, i.e., using larger ciphertext polynomials increases security, and using wider coefficients decreases it.

BitPacker, RNS-CKKS, and CKKS use the same N, and all are guaranteed to keep every ciphertext modulus Q smaller than Q_{max}, as defined by the program constraints. The exact moduli used by BitPacker, RNS-CKKS, and CKKS are different (since they use different representations), but the only relevant aspect for security is that they do not exceed Q_{max}.

Security in R-LWE also depends on other parameters, such as the sparsity of the secret key or the distribution of noise at encryption time [4]. But these aspects are independent of how ciphertexts are represented. Thus, BitPacker achieves the same level of security as other CKKS implementations.

4 BitPacker Makes FHE Accelerators Efficient

So far we have presented BitPacker's algorithms, with only a high-level description of why it helps performance. We now describe these benefits concretely. Sec. 4.1 reviews state-of-the-art accelerators and presents our baseline; Sec. 4.2 explains how BitPacker benefits performance, energy, and area; and Sec. 4.3 explains how BitPacker maps to accelerators.

4.1 Overview of CKKS accelerators

CraterLake [40], ARK [27], and SHARP [26] are the current state-of-the-art CKKS accelerators. CraterLake and ARK were developed concurrently, and have many similarities: they feature wide-vector functional units tailored to the primitive operations of FHE, include large and explicitly orchestrated on-chip memories to hold CKKS's enormous ciphertexts, and use compilers and algorithms that achieve high data reuse and high throughput. SHARP [27] builds on ARK, and incorporates techniques from CraterLake to further improve efficiency.

The key difference among these accelerators is their hardware word width. CraterLake uses narrow 28-bit words, so RNS-CKKS needs multiple residues per level, whereas ARK uses wider 64-bit words, and uses one residue per level. SHARP's key contribution is to optimize word size: SHARP uses RNS-CKKS with 36-bit words, and changes programs to use non-bootstrap scales that fit within 36 bits. SHARP's intermediate scale is more efficient than the prior 28-bit and 64-bit extremes. But as we will see, *RNS-CKKS is still inefficient across the range* of word widths, and BitPacker provides substantial savings over all these accelerators.

Earlier ASIC CKKS accelerators exist, but are not as fast: BTS [29] suffers from high memory traffic that limits performance; and F1 [39] is a smaller accelerator that cannot run programs with bootstrapping efficiently. Finally, FAB [2] and Poseidon [41] are recent FPGA accelerators that, though efficient, are slower than ASICs. All these accelerators use RNS-CKKS, so BitPacker would make them more efficient.

Baseline: CraterLake, ARK, and SHARP are similar at the hardware level, except for word size (see raw throughputs and areas in [26, Table 4]). However, they use different software stacks that make a direct comparison hard. Prior work [26, 27] compares them by using performance and energy reported by different papers, but this conflates major differences in benchmarks (e.g., using different CKKS scales and bootstrap algorithms) and prevents a controlled comparison.

To address this, we use CraterLake as the accelerator baseline, and sweep wordsize to create ARK-like (64-bit) and SHARP-like (36-bit) configurations. This allows comparing designs representative of these accelerators in a controlled way, with the same applications and optimizations.

Fig. 9 shows an overview of CraterLake. CraterLake is a 2048-lane vector processor. It has six types of vector functional units (FUs). Four of these are needed for functional completeness: modular adder and multiplier FUs perform element-wise vector operations; NTT FUs perform number-theoretic transforms, needed to multiply polynomials; and automorphism FUs implement structured permutations needed for homomorphic rotates. The other two units are performance optimizations: the CRB FU performs change-of-RNS-base operations, a common and costly operation that encapsulates many structured multiply-accumulates (ARK and SHARP have a similar FU, *bConv*); and the KSHGen FU generates some auxiliary data (keyswitch hints) on-chip to reduce memory traffic (ARK lacks this FU, but SHARP adopts it).

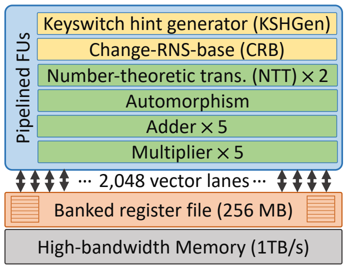
Pipelined FUs
Keyswitch hint generator (KSHGen)
Change-RNS-base (CRB)
Number-theoretic trans. (NTT) × 2
Automorphism
Adder × 5
Multiplier × 5
··· 2,048 vector lanes ···
Banked register file (256 MB)
High-bandwidth Memory (1TB/s)

Figure 9. CraterLake overview.

These accelerators spend most of their area and energy on computation. For example, in CraterLake, functional units take 50% of area and 60-80% of energy [40]. Moreover, the dominant component of functional units is multipliers, which take 70% of FU area in CraterLake. Thus, RNS-CKKS's inefficiency has significant impacts, as we will see below.

4.2 BitPacker improves performance, energy, and area

To understand BitPacker's benefits, we discuss how performance and energy change with the number of residues R. We focus on the cost of a homomorphic multiplication with $N = 64K$; homomorphic rotations have nearly identical costs, and homomorphic additions have negligible costs.

Performance analysis: A single homomorphic multiplication requires $O(R^2)$ multiplies, $O(R^2)$ adds, and $O(R)$ NTTs of residue polynomials (N elements each). NTTs are complex, requiring about 16× more energy than an element-wise multiply. To achieve high NTT utilization across values of R, the CRB unit encapsulates most multiplies and adds. Each CRB instruction performs $O(R)$ polynomial multiply-accumulates. Thus, each FU is used $O(R)$ times per homomorphic multiply.

Besides compute, memory also adds overheads: ciphertext size is linear with R, so by using smaller ciphertexts, BitPacker incurs fewer memory stalls than RNS-CKKS.

By combining compute and memory savings, BitPacker improves performance superlinearly. Since accelerators seek to balance compute and memory utilization, both effects are important. In practice, we observe that performance grows by about $R^{1.5}$ on the balanced systems we evaluate (Sec. 6).

Energy analysis: Fig. 10 shows a breakdown of energy per component, including different functional units and the register file (we assume all operands are on-chip; memory, which we include later, typically adds minor energy costs). The CRB and NTT FUs dominate energy, as they perform most

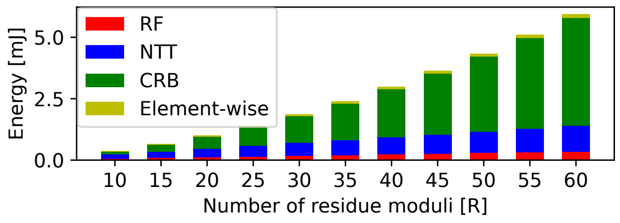

Figure 10. Energy breakdown for a homomorphic multiply as a function of residues R, for a 28-bit hardware word size.

operations. Overall, Fig. 10 shows that energy grows superlinearly with R, by about $O(R^{1.6})$. Growth is sub-quadratic because only the CRB grows quadratically; NTT and RegFile energy grow linearly. By reducing R, BitPacker reduces energy superlinearly.

Area analysis: For a given hardware configuration, two components can be reduced linearly without loss of performance when moving from RNS-CKKS to BitPacker: the CRB and the register file. The CRB is sized to perform R_{max} multiply-adds per input element, so by reducing R_{max}, we can reduce the number of multiply-adds per CRB lane without loss of performance. And since ciphertext size is proportional to R, by reducing R, the register file needs less space to hold the same number of ciphertexts. This yields significant savings, since FHE accelerators have large register files, e.g., taking 40% of die area in CraterLake [40, Table 2].

4.3 Mapping BitPacker level management to accelerators

BitPacker's rescale and adjust procedures (Sec. 3.2) take somewhat more computation than RNS-CKKS's. However, they can leverage the CRB FU in CraterLake (or the *bConv* FU in ARK and SHARP) to avoid a performance penalty.

Specifically, the main kernels are `scaleUp` and `scaleDown`. `scaleUp` is cheap: it multiplies all residues by a single, precomputed value. `scaleDown` has $2k(R-k)$ multiplies of residue polynomials, where k is the number of shed moduli. Thus, `scaleDown` can be 2-3× costlier than `scaleUp`.

Note that this overhead is minor compared to the $O(R^2)$ multiplies and adds per homomorphic multiply, as discussed above. Furthermore, `scaleDown`'s compute (lines 10–13) can be handled by the CRB, so in these CKKS accelerators, scaling down by k residues at a time is almost as fast as scaling down by a single one. As we will see, this makes BitPacker's level management costs comparable to RNS-CKKS's.

5 Methodology

Accelerators: We compare BitPacker and RNS-CKKS as described in Sec. 4.1. Our default accelerator is CraterLake as proposed, using 28-bit words, 2048 vector lanes, a 256MB register file, and 1 TB/s HBM memory. We then evaluate accelerator variants with word sizes from 28 to 64 bits. These capture the tradeoffs of word size, and are representative of ARK [27] (64-bit) and SHARP [26] (36-bit). We measure performance using CraterLake's cycle-accurate simulator [40]. We estimate performance and area using CraterLake's RTL implementation, which is synthesized in a commercial 12/14nm technology process using state-of-the-art tools [40]. We scale the area and energy fo CraterLake's components to evaluate different word sizes. We compute energy by combining per-component energies from RTL synthesis with cycle-level activity factors from CraterLake's simulator.

Benchmarks: We use five *large FHE programs* developed by FHE experts to evaluate BitPacker:

(1) ResNet-20 is Lee et al. [32]'s FHE implementation of the ResNet-20 deep neural network. ResNet-20 uses a high-degree polynomial to approximate ReLU activation functions, which is accurate but adds depth, making bootstrapping more frequent. It uses the CIFAR-10 dataset.

(2) ResNet-20+AESPA uses Park et al. [35]'s AESPA to improve the efficiency of ResNet-20. AESPA uses a low-degree (degree-2) polynomial in place of ReLU activation functions. This makes their implementation very low depth, reducing bootstrapping and execution time. The tradeoff is that, whereas ResNet-20 with ReLUs can use the same weights as the unencrypted network, ResNet-20+AESPA requires training to produce new weights. We follow AESPA's training procedure (using CIFAR-10 as in ResNet-20) and observe negligible impact on accuracy (achieving a 91.9% accuracy).

(3) RNN performs sentiment analysis using a Recurrent Neural Network [13]. It is derived from the LSTM benchmark in [40]. RNN processes 200 word embeddings x_i, and incorporates each in its hidden state following $h_{i+1} = \sigma(W_{hh}h_i + W_{ih}x_i + b)$. $\sigma(\cdot)$ is a degree-3 polynomial, and x_i and h_i are both of dimension 128. It uses the IMDB dataset [34].

(4) SqueezeNet is an FHE implementation of the SqueezeNet neural network [23, 30]. SqueezeNet uses degree-2 activation functions following AESPA [35], so it bootstraps less frequently than ResNet-20. It uses the CIFAR-10 dataset.

(5) LogReg uses the HELR algorithm [21] to perform logistic regression training. LogReg performs 32 iterations of Nesterov Accelerated Gradient Descent [38] with batch size 1024 and 197 features per sample. It uses the MNIST dataset [31].

These applications need different scales for application-level computation: ResNet and RNN use 45-bit scales, while SqueezeNet and LogReg use 35-bit scales.

Bootstrapping: All applications use bootstrapping. We use two state-of-the-art bootstrapping algorithms from Lattigo [1], which have different levels of end-to-end precision, 19 and 26 bits. We denote these algorithms *BS19* and *BS26*. BS26 is a bit costlier than BS19, but achieves higher precision.

These algorithms use different scales: *BS19* uses scales of 52, 55, and 30 bits; *BS26* uses scales of 54, 60, and 40 bits.

FHE parameters: We use ciphertext polynomials with $N = 64K$ coefficient and $\log_2 Q_{max} = 1,596$ bits. We use a combination of 3-digit, 2-digit, and 1-digit keyswitching [17, 20, 40]. These parameters achieve 128-bit security [40].

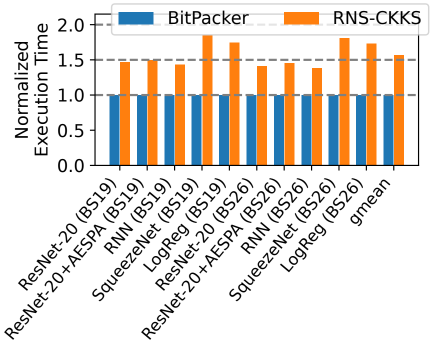

Figure 11. Execution time for BitPacker and RNS-CKKS on CraterLake with 28-bit words (lower is better).

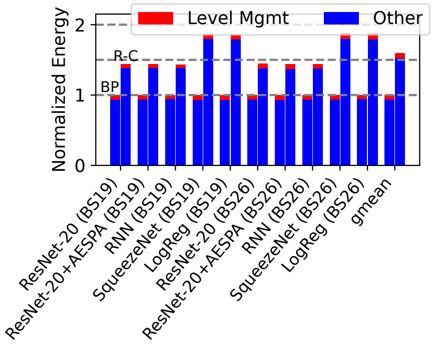

Figure 12. Energy for BitPacker (BP) and RNS-CKKS (R-C) on CraterLake with 28-bit words (lower is better).

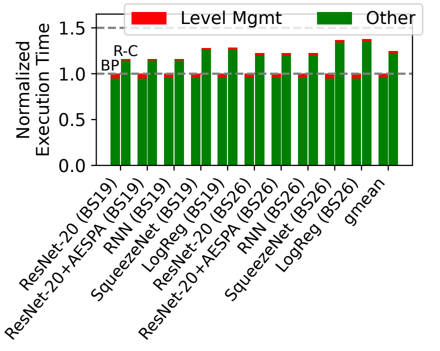

Figure 13. Execution time for BitPacker (BP) and RNS-CKKS (R-C) on a CPU with 64-bit words (lower is better).

As noted above, each application uses four diverse CKKS scales ranging from 30 to 60 bits: one for application computation, and three for bootstrapping.

Due to the lack of small NTT-friendly primes (Sec. 3.3), when hardware word sizes are narrow, there are some scales in the 30–35-bit range that RNS-CKKS cannot meet, as SHARP notes [26]. For example, with 28-bit words, a 30-bit scale is not possible, because there is no combination of two residue moduli whose bitwidth adds up to 30 bits. In these cases, we use the smallest possible scale. For instance, a single 35-bit scale is possible by combining 17- and 18-bit residue moduli. This consumes ciphertext bits more rapidly when scales are 35-bit or lower, but it is an unavoidable inefficiency of RNS-CKKS. Designs with 35-bit or larger word sizes do not suffer this problem, and this does not affect BitPacker, which can match any scale (BitPacker's only requirement is that enough NTT-friendly primes exist, which $w \geq 28$ bits meets).

BitPacker CPU implementation: We implement a single-threaded FHE library in Rust, similar to Lattigo, that supports BitPacker and RNS-CKKS. We use this library to evaluate BitPacker's accuracy and its performance on a 3.5 GHz AMD Zen 2 CPU (a Ryzen Threadripper PRO 3975WX).

6 Evaluation

6.1 BitPacker with 28-bit hardware words

Fig. 11 compares the execution time of BitPacker and RNS-CKKS across workloads, when using the default CraterLake configuration with 28-bit words (lower is better). BitPacker achieves a gmean 59% speedup over RNS-CKKS.

BitPacker helps all workloads, but speedups are higher on workloads that use smaller scales, which are less efficient in RNS-CKKS: SqueezeNet and LogReg, which use 35-bit scales for application work, see higher speedups than ResNet-20 and RNN, which use 45-bit scales; and BS19 variants see slightly higher speedups than BS26, since in RNS-CKKS, BS19's scales pack worse to 28-bit words.

Fig. 12 compares the energy consumed by 28-bit CraterLake when running BitPacker and RNS-CKKS, normalized to BitPacker (lower is better). Trends are similar to performance: BitPacker reduces energy over RNS-CKKS by gmean 59%. By improving performance *and* energy efficiency, BitPacker reduces energy-delay product (EDP) by 2.53× over RNS-CKKS (equivalently, improves performance/Joule by this amount).

Fig. 12 also breaks down energy consumption by level management operations (rescale and adjust), shown in **red**, vs. other operations, shown in **blue**. Level management costs are small for both BitPacker and RNS-CKKS (6% and 7% gmean, respectively). Interestingly, in absolute terms, BitPacker level management costs are lower than RNS-CKKS. This is because, while BitPacker switches multiple residues per level, it leverages the CRB to do so cheaply (Sec. 4.3), and at 28 bits, RNS-CKKS uses two residues per level, so it suffers from a larger number of residues.

Performance across FHE parameters: The above results achieve 128-bit security. We find similar benefits on FHE parameters that achieve 80-bit security: 53% gmean speedup (vs. 59% for 128-bit), and 63% lower energy (vs. 59% for 128-bit). This change stems from using lower-digit keyswitching for 80-bit security (2- and 1-digit) [40], which makes keyswitching take less time. Overall, BitPacker's benefits are similar because all parameters benefit from its more compact representation.

6.2 Comparison across word sizes

To characterize BitPacker's benefits more broadly, we evaluate accelerator variants with different hardware word sizes. We evaluate designs with 28- to 64-bit words. Since changing word size has several complex effects, we perform *iso-throughput* scaling: as we increase word size, we proportionately *reduce the number of vector lanes*, so that the total number of bits consumed per cycle stays roughly constant. For example, the 30-bit accelerator has twice the vector lanes of the 60-bit accelerator (we do this scaling by changing CraterLake's lane groups [40]). All designs use the same register file size and memory bandwidth. We retain the same balance of functional units, except that wider words reduce R_{max}, so

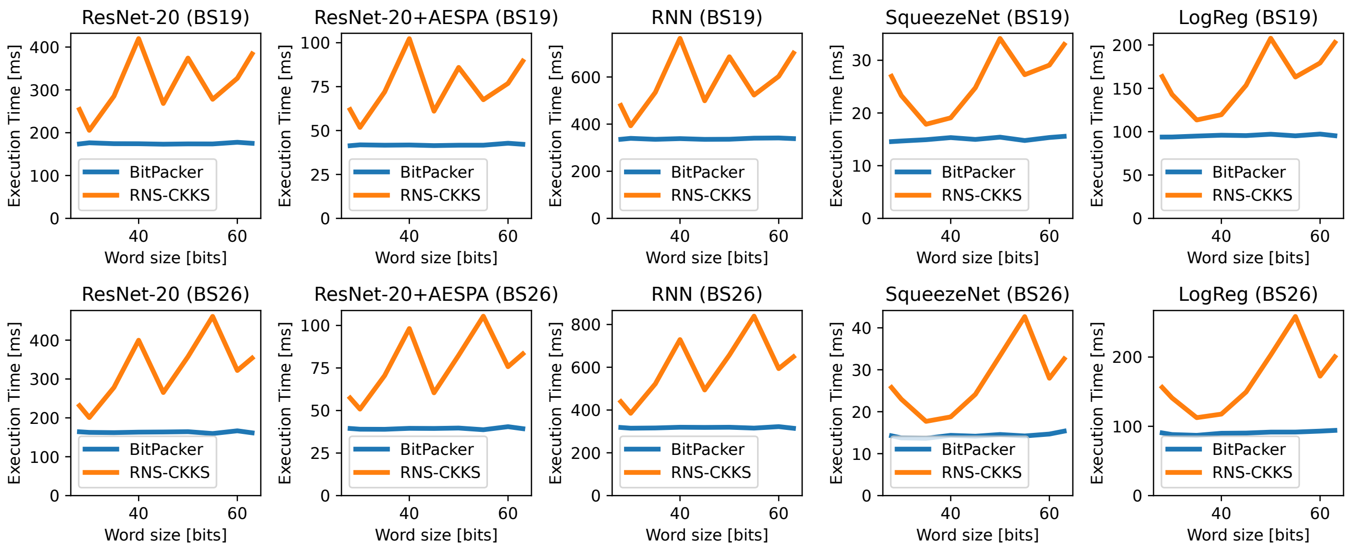

Figure 14. Execution time in milliseconds on CraterLake of BitPacker compared to RNS-CKKS as word size changes.

we reduce the number of multiply-adds per lane of the CRB unit linearly (otherwise, the CRB would be overdesigned). For example, the 30-bit design has a CRB with 56 MAC units per lane, whereas the 60-bit design uses 28 per lane.

Overall, this scaling strategy *maintains the same raw computational throughput across word sizes*. But designs with wider words have more area, because multipliers scale quadratically. Specifically, whereas the 28-bit design takes 472mm^2 in a commercial 14/12nm process [40], the 64-bit design consumes 557mm^2, chiefly due to a larger NTT unit.

Fig. 14 shows how the execution time (y-axis) of BitPacker and RNS-CKKS varies as we scale word size from 28 to 64 bits (x-axis). Each plot shows results for a different application. Fig. 14 shows a stark difference in behaviors. BitPacker maintains *constant performance across word sizes*, because it fully leverages the accelerator's raw throughput, which is constant given our scaling methodology.

Fig. 14 also shows that RNS-CKKS always performs worse than BitPacker. RNS-CKKS has *extremely uneven* performance across word sizes, with peaks and valleys that are about 2× apart. The valleys correspond to points where the word size matches one of the scales in the program. For example, ResNet-20 with BS19 bootstrapping uses scales of 45 bits (outside of bootstrapping) and 30, 52, and 55 bits (during bootstrapping). The valleys occur at these points because RNS-CKKS matches residue sizes to scales, and some fraction of the residues (but never all) achieve good datapath utilization. Conversely, peaks happen at word sizes that cause poor utilization across the board, like 40 bits in ResNet-20.

Since different applications use different scales, Fig. 14 shows that each benchmark shows different peaks and valleys, so the optimal word size for RNS-CKKS varies significantly across applications. BitPacker erases this problem, providing uniformly high utilization for any scale.

To better capture performance trends, Fig. 15 shows the gmean, maximum, and minimum slowdowns of RNS-CKKS vs. BitPacker across all benchmarks. This again highlights that RNS-CKKS is inefficient across word sizes, and shows that large word sizes are somewhat more affected by poor utilization. For example, at 64 bits (ARK-like configuration), RNS-CKKS's gmean slowdown is 2.18×, vs. 59% at 28 bits.

Finally, Fig. 16 shows BitPacker and RNS-CKKS execution times by unit area, relative to the execution time × area ratio of BitPacker with 28-bit words (this is the inverse of performance per area). Recall that, to provide the same raw throughput, designs with wider words use somewhat more area: the 64-bit accelerator is 18% larger than the 28-bit one. This is why the BitPacker line now trends upwards, and the RNS-CKKS line grows more quickly than in Fig. 15. Overall, RNS-CKKS at 64 bits (ARK-like configuration) has 2.5× worse performance/area than BitPacker at 28 bits. This shows that narrower word sizes are more efficient, and that BitPacker with 28-bit words is the most efficient choice.

Comparison with SHARP: The 36-bit points in Figs. 14–16 capture the key contribution of SHARP [26], tuning word size. However, BitPacker has multiple advantages over SHARP.

First, SHARP requires changing applications to use scales close to 36 bits, whereas BitPacker works with arbitrary scales. Using 36-bit scales limits *fixed-point* precision to about 20 bits, which is insufficient for many applications (this is a far smaller range than 16-bit *floating-point*, often used in neural networks). SHARP evaluated 3 applications; ResNet-20 needed simple changes to work with smaller scales, and sorting suffered added error. Three of our workloads (ResNet-20, ResNet-20+AESPA, and RNN) use 45-bit scales and would require two primes per level in SHARP, which would hinder efficiency. We found that RNN diverged when using 35-bit scales; for ResNet-20, we use 45-bit scales following Lee

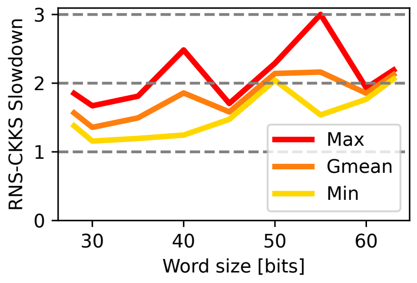

Figure 15. Gmean, maximum, and minimum slowdown for RNS-CKKS compared to BitPacker across word sizes.

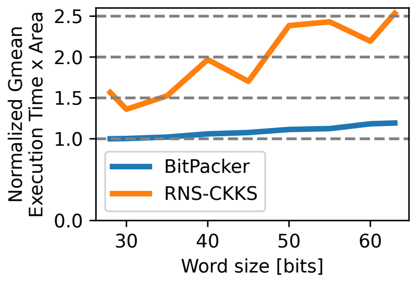

Figure 16. Gmean inverse performance per unit area across word sizes for BitPacker and RNS-CKKS (lower is better).

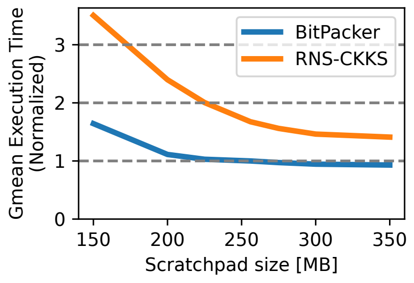

Figure 17. Gmean execution time on 28-bit CraterLake across register file sizes, relative to BitPacker at 256 MB.

et al.'s implementation [32], but we were able to use 35-bit scales with minimal loss of accuracy by applying the changes discussed in SHARP, replicating their result.

Second, SHARP hinders bootstrapping, which needs a wide range of scales. SqueezeNet and LogReg have 35-bit application scales, well-matched to SHARP, but Fig. 14 shows significant overheads, about 30%, due to bootstrapping costs.

Third, BitPacker enables using a 28-bit datapath, which is more efficient than a 36-bit one, as we have seen.

As a result, BitPacker at 28-bit words is gmean 43% faster than the 36-bit SHARP-like design and has 2.2× better EDP.

6.3 BitPacker reduces accelerator area

So far, we have evaluated accelerators that were tuned for RNS-CKKS. But BitPacker uses fewer residues, so we can reduce area without hurting performance, as Sec. 4.2 explained.

First, Fig. 17 shows gmean execution time as the accelerator's register file size (x-axis) changes. Results are normalized to BitPacker at 256 MB. RNS-CKKS plateaus at 256 MB but steadily loses performance at lower sizes. By contrast, BitPacker sees no loss in performance from 256 MB until 200 MB. Even at 150 MB, BitPacker sees only a 70% slowdown, while RNS-CKKS slows down by over 3×. Second, as Sec. 4.2 discusses, we find that we can make CraterLake's CRB 28% smaller with no performance regression for BitPacker.

Thus, BitPacker enables using a CraterLake configuration that has 395.5 mm^2 instead of the original 472.3 mm^2, with no loss in performance, a 19% area reduction.

Combining BitPacker's performance, energy, and area improvements, we see that BitPacker with this configuration improves energy-delay-area product by 3.0× over RNS-CKKS on the original CraterLake configuration.

6.4 BitPacker performance on CPUs

Fig. 13 compares the performance of BitPacker and RNS-CKKS on a CPU. We use 64-bit words, the best choice on CPUs (32-bit words have 2× vector throughput, but over 2× instructions per homomorphic operation). BitPacker is gmean 24% faster than RNS-CKKS; this is significant, but lower than the 2.1× speedup on 64-bit accelerators (Fig. 15).

	Mean		Worst-case	
Benchmark	BitPacker	RNS-CKKS	BitPacker	RNS-CKKS
ResNet-20	20.7	20.8	18	18
ResNet-20+AESPA	12.8	13.8	8	9
RNN	23.3	23.2	22	22
SqueezeNet	15.4	15.0	13	13
LogReg	11.7	11.7	9	9

Table 1. Error-free mantissa bits in BitPacker and RNS-CKKS.

The speedup is lower because, due to the lack of specialized FUs, NTTs (which grow with R) dominate and **level management** is a bit costlier; and due to the CPU's limited compute, memory isn't a bottleneck. Thus, while BitPacker is broadly beneficial, it helps accelerators much more.

6.5 BitPacker preserves accuracy

BitPacker improves performance without compromising precision. We first compare BitPacker's accuracy with RNS-CKKS in end-to-end applications, and then analyze the accuracy of individual level management operations, rescale and adjust.

Table 1 shows that BitPacker matches the precision of RNS-CKKS across workloads. We run many samples to ensure precision is stable (all results have 95^{th} percentile confidence intervals below 1%). RNS-CKKS precision results use 64-bit words because RNS-CKKS has better precision with 64-bit words than with smaller word sizes [25]. BitPacker precision results use 28-bit words, the choice that limits residue moduli the most and thus has the highest risk to impact precision; we observe no accuracy changes with larger word sizes.

Table 1 reports mean and worst-case error for each benchmark by comparing the outputs of each sample with those of unencrypted computation using 64-bit floating-point values. Table 1 reports the number of error-free mantissa bits. For all applications except ResNet-20+AESPA, mean error differences are small and well within the 0.5-bit margin that we set for choosing moduli, and worst-case errors all match.

For ResNet-20+AESPA, Table 1 shows that BitPacker's mean and worst-case errors are 1 bit worse than RNS-CKKS.

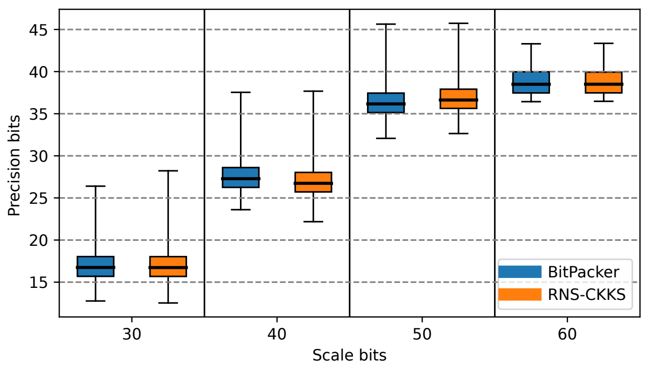

Figure 18. Comparison of error distributions for the *rescale* operation, for 28-bit BitPacker (BP) and 64-bit RNS-CKKS (R-C). Box-and-whisker plots show the distribution of error-free mantissa bits (the box denotes quartiles, and whiskers show min/max values).

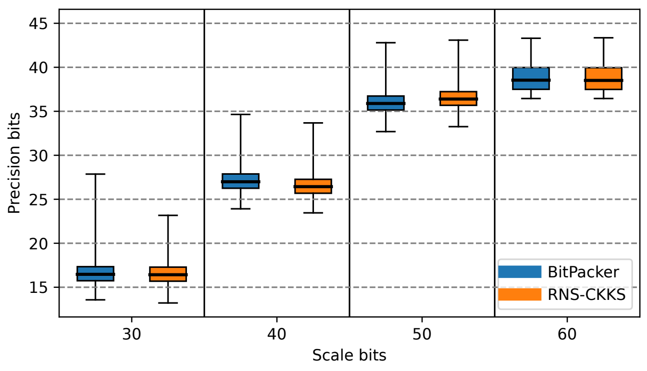

Figure 19. Comparison of error distributions for the *adjust* operation, for 28-bit BitPacker (BP) and 64-bit RNS-CKKS (R-C). Box-and-whisker plots show the distribution of error-free mantissa bits (the box denotes quartiles, and whiskers show min/max values).

Table 1 also shows that numerical errors are higher than the other applications that use 45-bit scales (ResNet-20 and RNN have >20 error-free mantissa bits; ResNet-20+AESPA has 13.8). This happens because ResNet-20+AESPA is less numerically stable and quite sensitive to scale: with RNS-CKKS, reducing scale by 1 bit (from 45 to 44 bits) reduces the average number of error-free mantissa bits by 2.4, from 13.8 to 11.4 bits. BitPacker's 1-bit difference happens because its moduli selection algorithm often chooses slightly smaller scales (within 0.5 bits of the target). Despite this difference, RNS-CKKS and BitPacker always produce the same classifications and achieve the same end-to-end accuracy, 91.9%.

Since BitPacker's key contribution is a new approach to level management, we analyze the accuracy of these operations in more depth. We follow a similar methodology to Kim et al. [25]: for rescale, we measure the statistical distribution of error after squaring and rescaling ciphertexts with values uniformly distributed in $[-1, 1]$; for adjust, we measure error after adjusting by one level. In both cases, we use ciphertexts with a starting level L =10, and use scales ranging from 30 to 60 bits. Like before, we compare BitPacker with 28-bit words (the most limiting choice) and RNS-CKKS using 64-bit words. To ensure statistical significance, we use 1 million samples.

Fig. 18 and Fig. 19 report the distribution of errors for rescale and adjust, respectively. Each distribution is reported as a *box-and-whiskers* plot, where the box denotes the quartiles (i.e., the bottom of the box is the 25^{th} percentile, the middle line is the median, and the top of the box is the 75^{th} percentile), and the whiskers report the minimum and maximum values. Each group of two box plots reports results for a different scale. Precision is given in bits (i.e., $-log_2(error)$).

Fig. 18 and Fig. 19 show that BitPacker's level management operations have error distributions with negligible differences from those of RNS-CKKS. Like before, these differences are within the 0.5-bit margin we set when selecting BitPacker moduli.

7 Conclusion

Making full use of FHE accelerators requires redesigning FHE algorithms to use them well. We have presented BitPacker, a new implementation of the state-of-the-art CKKS scheme that addresses the overheads of standard RNS-CKKS. BitPacker decouples CKKS scales from RNS residues, enabling high utilization of accelerator datapaths. This significantly improves performance, efficiency, and area, with no precision loss.

Acknowledgments

We thank the anonymous reviewers; our shepherd, Jung Ho Ahn; and Aleksandar Krastev, Axel Feldmann, Hyun Ryong Lee, Quan Nguyen, Yifan Yang, Victor Ying, Shabnam Sheikhha, Fares Elsabbagh, and Joel Emer for feedback on earlier versions of this manuscript. We thank Aleksandar Krastev for help with benchmarks. This research was funded in part by a Wistron research grant.

References

[1] 2020. Lattigo. https://github.com/ldsec/lattigo.
[2] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. 2023. FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption. In *HPCA-29*.
[3] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. 2022. OpenFHE: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography.*
[4] Martin R Albrecht, Benjamin R Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W Postlethwaite, Fernando Virdia, and Thomas Wunderer. 2018. Estimate all the {LWE, NTRU} schemes!. In *Proc. of the 1tth int. conf. on Security and Cryptography for Networks (SCN).*
[5] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. 2017. A full RNS variant of FV like somewhat homomorphic encryption schemes. In *Proc. of the 23rd intl. conf. on Selected Areas in Cryptography (SAC).*

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

[7] Zvika Brakerski and Vinod Vaikuntanathan. 2011. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Proc. of the 31st Annual Cryptology Conference (CRYPTO).*

[8] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS).*

[9] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. 2018. A full RNS variant of approximate homomorphic encryption. In *Proc. of the 25th intl. conf. on Selected Areas in Cryptography (SAC).*

[10] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT).*

[11] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in less than 0.1 Seconds. In *ASIACRYPT.*

[12] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *EUROCRYPT.*

[13] Jeffrey L Elman. 1990. Finding structure in time. *Cognitive science* 14, 2 (1990).

[14] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.* (2012).

[15] Harvey L Garner. 1959. The residue number system. In *Papers presented at the the March 3-5, 1959, Western Joint Computer Conference.*

[16] Craig Gentry and Shai Halevi. 2019. Compressible FHE with Applications to PIR. In *Proceedings of the Theory of Cryptography Conference (TCC).*

[17] Craig Gentry, Shai Halevi, and Nigel P Smart. 2012. Homomorphic evaluation of the AES circuit. In *Proceedings of the 32nd Annual Cryptology Conference (CRYPTO).*

[18] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proc. of the Intl. Conf. on Machine Learning (ICML).*

[19] Shai Halevi and Victor Shoup. 2020. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Paper 2020/1481. https://eprint.iacr.org/2020/1481

[20] Shai Halevi and Victor Shoup. 2020. *HElib design principles.* Technical Report.

[21] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2018. Efficient Logistic Regression on Large Encrypted Data. Cryptology ePrint Archive, Report 2018/662.

[22] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. 2019. Logistic regression on homomorphic encrypted data at scale. In *Proc. of the AAAI Conference on Artificial Intelligence*, Vol. 33.

[23] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[24] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A low latency framework for secure neural network inference. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18).*

[25] Andrey Kim, Antonis Papadimitriou, and Yuriy Polyakov. 2022. Approximate homomorphic encryption with reduced approximation error. In *Proc. RSA Conference, Cryptography Track (CT-RSA).*

[26] Jongmin Kim, Sangpyo Kim, Jaewan Choi, Jaiyoung Park, Donghwan Kim, and Jung Ho Ahn. 2023. SHARP: A Short-Word Hierarchical Accelerator for Robust and Practical Fully Homomorphic Encryption. In *ISCA-50.*

[27] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. 2022. ARK: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse. In *MICRO-55.*

[28] Miran Kim, Yongsoo Song, Baiyu Li, and Daniele Micciancio. 2020. Semi-parallel logistic regression for GWAS on encrypted data. *BMC Medical Genomics* 13, 7 (2020).

[29] Sangpyo Kim, Jongmin Kim, Michael Jaemin Kim, Wonkyung Jung, John Kim, Minsoo Rhu, and Jung Ho Ahn. 2022. BTS: An accelerator for bootstrappable fully homomorphic encryption. In *ISCA-49.*

[30] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images.* Technical Report. University of Toronto.

[31] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998).

[32] Eunsang Lee, Joon-Woo Lee, Junghyun Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and Woosuk Choi. 2022. Low-Complexity Deep Convolutional Neural Networks on Fully Homomorphic Encryption Using Multiplexed Parallel Convolutions. In *Proc. of the Intl. Conf. on Machine Learning (ICML).*

[33] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2013. On ideal lattices and learning with errors over rings. *Journal of the ACM (JACM)* 60, 6 (2013).

[34] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proc. of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies.*

[35] Jaiyoung Park, Michael Jaemin Kim, Wonkyung Jung, and Jung Ho Ahn. 2022. AESPA: Accuracy preserving low-degree polynomial activation for fast private inference. *arXiv preprint arXiv:2201.06699* (2022).

[36] Chris Peikert. 2016. A decade of lattice cryptography. *Foundations and Trends in Theoretical Computer Science* 10, 4 (2016).

[37] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. 2020. HEAX: An architecture for computing on encrypted data. In *ASPLOS-XXV.*

[38] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[39] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. 2021. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54.*

[40] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. 2022. CraterLake: A hardware accelerator for efficient unbounded computation on encrypted data. In *ISCA-49.*

[41] Yinghao Yang, Huaizhi Zhang, Shengyu Fan, Hang Lu, Mingzhe Zhang, and Xiaowei Li. 2023. Poseidon: Practical Homomorphic Encryption Accelerator. In *HPCA-29.*