Fine-Grained and Runtime-Configurable Precision for Exact FHE Inference

Wun-Ting Lin and Ja-Ling Wu

National Taiwan University, Taipei, Taiwan
d07922020@ntu.edu.tw,wjl@cmlab.csie.ntu.edu.tw

Abstract. Privacy-preserving machine learning under fully homomorphic encryption (FHE) faces a structural limitation: numerical precision is bound to cryptographic parameters and key material, forcing precision to be fixed at scheme initialization. Existing frameworks must regenerate keys or recompile circuits whenever bit-width changes, eliminating precision as a deployment-time performance knob and making mixed-precision strategies—widely used in plaintext machine learning—impractical under encryption.

We present Tailor, a backend-agnostic framework for exact, runtime-configurable mixed-precision neural inference over bitwise FHE. By representing signed integers as vectors of independently encrypted bits and constructing all arithmetic and neural-network operators from precision-parametric Boolean circuits, Tailor decouples bit-width from cryptographic state. Per-layer precision becomes a runtime parameter under a single keygen, and nonlinear operators, including ReLU, absolute value, and comparisons, are evaluated exactly rather than via polynomial approximation. A fused saturating requantize–ReLU produces compact unsigned activations, and each accumulator is provisioned at its provably minimal width. A `Scheme<Backend>` abstraction makes the framework portable across bitwise FHE families; we instantiate it for both TFHE and FINAL, along with a plaintext reference backend for functional validation and exact gate accounting.

Our evaluation of an MNIST classifier shows that mixed-precision configurations strictly Pareto-dominate uniform deployment. Reducing only the output-layer weights to 2 bits while keeping inputs, activations, and hidden-layer weights at 4 bits yields the highest accuracy among all evaluated settings, 96.44% in 222.0 s. This is both more accurate and faster than uniform 4-bit inference (96.17% in 236.1 s) and 2.8× faster than uniform 8-bit (96.14% in 632.4 s). Reducing the hidden-layer weights to 2 bits instead trades 0.4 accuracy points for a 1.5× speedup over uniform 4-bit (95.80% in 158.2 s).

Keywords: Fully Homomorphic Encryption · Privacy-Preserving Machine Learning · Mixed-Precision Inference

1 Introduction

Deep learning has become a central component of modern data-driven services, from image recognition to large-scale language models. Deploying these models often requires users to outsource computation to untrusted cloud providers, raising serious concerns about data privacy. Fully Homomorphic Encryption (FHE) addresses this challenge by enabling computation directly over encrypted data, allowing inference without revealing sensitive inputs.

Recent research on privacy-preserving machine learning (PPML) has produced a variety of FHE-based inference frameworks. However, whether they rely on arithmetic or bitwise constructions, most existing systems share a critical structural limitation: *numerical*

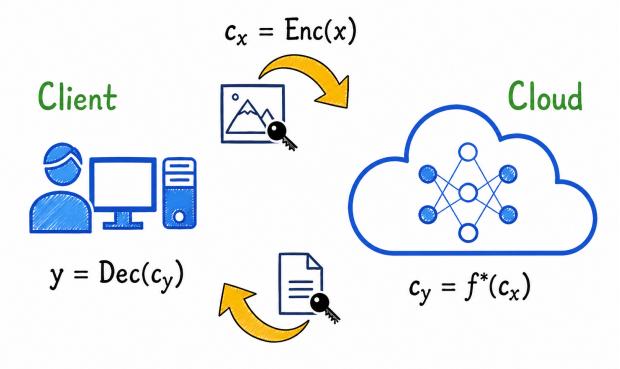


Figure 1: The privacy-preserving inference scenario

precision is inherently bound to the scheme's cryptographic parameters or compile-time state. Specifically, arithmetic FHE schemes [BGV12, Bra12, FV12, CKKS17] offer efficient homomorphic operations but rely on polynomial approximations for nonlinear functions, introducing numerical error and limiting exactness. Conversely, bitwise and recent NTRU-based schemes [DM14, CGGI16, BIP+22, XZD+23, JMPP24] evaluate nonlinear functions exactly via Boolean gates. Yet, frameworks built upon them—most notably Concrete-ML [Zam22a]—still bind precision to circuit compilation, requiring expensive re-keying or recompilation whenever the bit-width changes. This rigidity makes it difficult to exploit precision as a performance–accuracy trade-off knob, a technique widely used in plaintext machine learning systems.

This work poses a simple yet fundamental question: can precision be treated as a runtime-configurable system parameter in FHE-based inference, without modifying cryptographic parameters or keys?

Answering this question requires resolving three coupled challenges:

1. *Backend independence.* Bitwise FHE backends differ substantially in bootstrapping algorithms, key formats, and noise management. A framework that hard-codes a specific backend cannot reuse its arithmetic and neural-network logic across the evolving FHE landscape.

2. *Runtime precision under a fixed key.* Existing works treat bit-width as a compile-time or keygen-time parameter, forcing developers to regenerate keys each time precision is tuned. Decoupling precision from cryptographic state requires rethinking the implementation of integer arithmetic, neural operators, and tensor abstractions.

3. *Bridging trained models and encrypted execution.* Quantization-aware training (QAT) toolchains such as Brevitas [FPF25] target hardware accelerators under assumptions (unsigned activations, floating-point scale factors) that do not align with FHE's bit-sliced integer representation. A practical framework must close this representational gap.

We answer this question affirmatively by presenting Tailor, a backend-agnostic execution framework for exact, runtime-configurable mixed-precision FHE inference. Tailor is instantiated on TFHE [CGGI18], FINAL [BIP+22], and a plaintext gate-counting reference backend. It supports per-layer precision with a single keygen and integrates with PyTorch via a Brevitas-based QAT pipeline. We target the common PPML deployment scenario shown in Figure 1: encrypted user inputs are evaluated against plaintext model weights held by a semi-honest server. Under this threat model, Tailor provides bit-exact correctness while exposing precision as a runtime knob for applications.

On MNIST, every point on the resulting accuracy–latency Pareto frontier is achieved by TAILOR's low- or mixed-precision configurations, represented as per-layer bit-width tuples $(k_{\text{in}}, k_{w1}, k_{\text{act}}, k_{w2})$ covering the input, first-layer weights, activations, and second-layer weights: $(4, 4, 4, 2)$ attains the best accuracy (96.44%) and runs faster than both uniform 4-bit and uniform 8-bit execution (222.0 s vs. 236.1 s and 632.4 s). $(4, 2, 4, 4)$ trades 0.4 accuracy points for a further 1.5× speedup (158.2 s), demonstrating that runtime precision selection unlocks performance regimes inaccessible to uniform-precision deployment. Switching between precision configurations requires no keygen and no recompilation, in contrast to Concrete-ML and SEAL baselines, which incur reconfiguration costs of seconds.

1.1 Contributions

This paper makes the following contributions:

1. **Backend-agnostic execution architecture.** We introduce a unified abstraction layer that cleanly decouples bitwise FHE primitives from higher-level arithmetic and neural-network logic. By instantiating this layer on TFHE, FINAL plus a plaintext reference, we demonstrate native portability across LWE- and NTRU-based backends.

2. **Runtime-configurable exact arithmetic.** We design a comprehensive suite of homomorphic operators (e.g., addition, multiplication, ReLU) as bit-width-parametric circuits. This enables dynamic, k-bit precision scaling at runtime without altering the underlying cryptographic parameters. Layers execute with analytically minimal accumulator widths and a fused ReLU for unsigned activations.

3. **FHE-aware quantization and deployment pipeline.** We bridge the representational gap between standard QAT (Brevitas) and bit-sliced FHE. Our modified quantizers guarantee bit-identical round-trip inference, exposing per-layer precision to developers as a straightforward Python parameter.

4. **Security analysis and reconfiguration-cost evaluation.** Beyond standard threat modeling, we analyze the side-channel leakage of runtime precision selection. We also explicitly benchmark reconfiguration overheads against baselines like Concrete-ML and SEAL—a critical deployment dimension often overlooked in prior literature.

5. **End-to-end empirical validation.** We evaluate the framework on MNIST and CIFAR-10, measuring backend performance under identical workloads, including end-to-end encrypted inference across all three backend instantiations. We also validate an analytical gate-count cost model that accurately predicts performance scaling across larger network architectures.

2 Background and Related Work

This section reviews prior work most relevant to TAILOR: cryptographic backends that enable bitwise FHE, neural-network inference systems built on FHE, and the quantization techniques TAILOR relies on to exploit runtime-configurable precision. We conclude by positioning TAILOR relative to these prior efforts.

2.1 Bitwise FHE Backends

Unlike arithmetic FHE schemes that rely on polynomial approximations, bitwise FHE schemes evaluate computations using bootstrapped Boolean gates. This enables the exact evaluation of arbitrary non-linear functions—a critical requirement for accurate neural network inference.

Within the bitwise paradigm, two primary cryptographic foundations have emerged. The first is based on the Learning with Errors (LWE) assumption, dominated by FHEW [DM14] and TFHE [CGGI16], which evaluate bootstrapped gates in milliseconds. The second comprises recent NTRU-based constructions, most notably FINAL [BIP+22] and its extensions [XZD+23, JMPP24]. These offer compact ciphertext representations and competitive bootstrapping performance under conservative parameter choices.

Tailor is architected to be backend-agnostic, treating both LWE- and NTRU-based schemes as interchangeable components via a uniform Boolean-gate interface. We instantiate the framework on both TFHE and FINAL to demonstrate this portability and to quantitatively evaluate the latency trade-offs between the two backend families.

2.2 FHE-Based Private Neural Inference

A rich ecosystem of compilers and frameworks has emerged to bridge deep learning and FHE [WZB+23]. Within the arithmetic paradigm, pioneer systems like CryptoNets [DGBL+16], nGraph-HE2 [BCCW19] and subsequent compilers—such as EVA [DKS+20], CHET [DSC+19], and HELayers [AAB+23]—have successfully automated ciphertext optimization and SIMD packing. However, to accommodate their underlying schemes, these frameworks must adapt model architectures (e.g., replacing ReLU with square activations) or employ polynomial approximations, inherently trading exactness for throughput.

For applications demanding exact inference, bitwise frameworks are the standard. Concrete-ML [Zam22a] is the most prominent example, providing a scikit-learn–compatible interface atop TFHE [CGGI18]. It compiles quantized models by heavily utilizing programmable bootstrapping (PBS) to evaluate neural operators via small lookup tables. While highly optimized for fixed, low-bit quantizations, this PBS-centric design is precisely what binds precision to compile-time circuit specialization. Other approaches, such as HT2ML [WZB+23], bypass these limitations by offloading non-linear layers to Trusted Execution Environments (TEEs), though this trades away pure cryptographic guarantees.

Tailor diverges from these systems at the architectural level. Rather than relying on PBS-based lookup tables or hardware enclaves, it constructs all neural operators directly from precision-parametric Boolean circuits. This specific design choice allows bit-width to exist purely as a software-level parameter.

2.3 Quantization for Neural Networks

Quantization reduces the numerical precision of weights and activations to accelerate inference and reduce memory footprint. Post-training quantization (PTQ) calibrates a trained model using a small dataset, whereas quantization-aware training (QAT) incorporates quantization effects into the training loop and typically achieves higher accuracy at low bit-widths. Brevitas [FPF25], the QAT toolchain we build on, provides PyTorch-integrated quantizers and supports flexible per-layer bit-width configuration.

Mixed-precision quantization further improves the accuracy–efficiency trade-off by assigning varying bit-widths to different layers based on sensitivity analysis. While this technique is well established in plaintext ML systems to maximize hardware efficiency, it has remained largely unexploited in FHE due to the rigid precision bindings of existing frameworks. Tailor resolves this structural friction, allowing established mixed-precision strategies to be seamlessly deployed onto encrypted inference workloads.

2.4 Positioning of Tailor

Table 1 summarizes how Tailor differs from representative FHE inference systems across four dimensions: *exactness* (whether nonlinear operations introduce approximation error),

Table 1: Positioning of TAILOR against representative FHE inference systems.

System	Exact evaluate	Runtime precision	Multi-backend	Mixed precision
CryptoNets	✗	✗	✗	✗
nGraph-HE2	✗	✗	✗	✗
Concrete-ML	✓	✗	✗	partial
Tailor	✓	✓	✓	✓

runtime precision (whether bit-width can be reconfigured without re-keying), *backend diversity* (whether the framework abstracts multiple FHE backends), and *mixed-precision support* (whether per-layer bit-width can be selected at deployment).

To our knowledge, TAILOR is the first FHE inference framework that simultaneously provides bit-exact nonlinear evaluation, runtime-configurable mixed precision with a single keygen, and demonstrated portability across LWE- and NTRU-based backends.

3 Threat Model and Security Analysis

This section formalizes the deployment scenario, characterizes the security guarantees inherited from the underlying bitwise FHE backends, and analyzes residual leakage introduced by the selection of runtime precision.

3.1 System and Adversary Model

We adopt the standard client–server inference model for FHE [DGBL+16, BCCW19]. A client encrypts a private input x using a local secret key and sends it to a server, which homomorphically evaluates a plaintext model f to return $\mathsf{Enc}(f(x))$. We assume a computationally bounded, *semi-honest* server that faithfully executes the protocol but may attempt to infer x from ciphertexts, intermediate states, and timing.

Our primary security guarantee is *input confidentiality*: the adversary learns nothing about the plaintext x beyond standard metadata (e.g., tensor dimensions, model architecture, execution time). While TAILOR unconditionally guarantees bit-exact correctness, verifiable execution against a malicious server remains outside our current scope (Section 3.4).

3.2 Security of Bit-Sliced Integer Encryption

TAILOR represents each k-bit signed integer as a vector of k independently encrypted bits $\mathsf{Enc}(x_0), \ldots, \mathsf{Enc}(x_{k-1})$, where each bit is encrypted using the backend's encryption procedure. Therefore, TAILOR's IND-CPA security directly reduces to the backend's IND-CPA security.

Both backends used in this work are instantiated at the standard $\lambda = 128$-bit security level. TFHE relies on the hardness of LWE and FINAL relies on the NTRU assumption. Each bootstrapped Boolean gate refreshes ciphertext noise to a bounded level, thereby preserving IND-CPA security across arbitrary circuit depths.

A natural concern is whether encrypting an integer as k independent ciphertexts—rather than as a single packed ciphertext—introduces additional leakage. We argue this is not the case: each bit ciphertext is sampled with independent randomness, the backend's IND-CPA security ensures that no individual ciphertext leaks its plaintext bit. A hybrid argument over k bits yields standard multi-message IND-CPA security for composite integer encryption. The decomposition is structurally analogous to packed schemes that

encode multiple slots within a single ciphertext; the security argument differs only in the granularity of the encrypted units.

3.3 Leakage from Runtime Precision

Treating precision k as a software-level parameter introduces specific side channels that TAILOR explicitly bounds.

Precision is public We treat each layer's bit-width configuration as *public information* available to both the client and the server. Inference latency, memory consumption, and the number of issued homomorphic gates all scale with k, and any of these quantities is observable to a co-located or network-level adversary. TAILOR therefore does not attempt to hide the precision schedule. Applications that require precision secrecy can pad computation to a uniform worst-case bit-width at the corresponding performance cost; however, this is outside our default deployment model.

Data-dependent timing in plaintext-scalar operations TAILOR's scalar-multiplication optimization exploits the Hamming weight of plaintext weights and skips zero coefficients in linear layers. Because weights belong to the server in our deployment model, this timing variation reveals information about the server's own model rather than the client's input, and therefore does not violate input confidentiality. In deployments where model weights themselves are considered sensitive, this channel must be closed by replacing sparse and Hamming-weight–aware routines with constant-time variants. We flag this as a deployment-time configuration choice rather than a fundamental limitation.

Precision-induced timing distinguishers If a client issues queries under different precision schedules across sessions, the resulting latency differences allow a network observer to distinguish those sessions. This does not leak input content but does leak the client's precision preference. For applications where precision selection itself is sensitive, the client can adopt a fixed precision schedule, accepting the latency cost in exchange for indistinguishability across queries.

3.4 Scope and Non-Goals

We make the following limitations explicit:

Model weight confidentiality TAILOR targets input confidentiality; model weights are assumed to be the server's plaintext property. Protecting weights against a curious client would require encrypting weights as well and is orthogonal to our contribution.

Verifiability TAILOR guarantees bit-exact correctness only against a semi-honest server. A malicious server can substitute or omit computations and return ill-formed ciphertexts; the client has no built-in mechanism to detect such behavior. Verifiable FHE inference [VKH23, KHSS22] is an active research direction that can be composed with TAILOR but is not addressed here.

Metadata leakage Tensor shapes, layer counts, and the overall computation graph structure are visible to the server, as in essentially all FHE inference frameworks. Hiding them requires oblivious execution techniques, which are outside our scope.

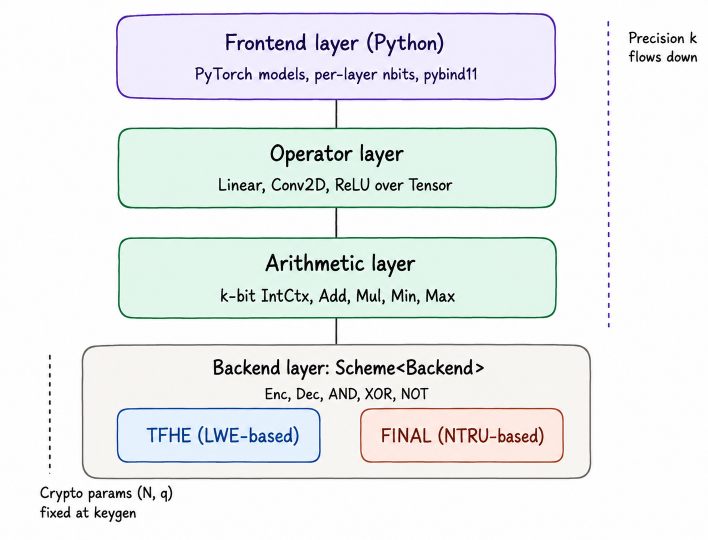


Figure 2: TAILOR architecture.

Plaintext reference backend The PLAIN backend (Section 4.2) evaluates the same operator graph over cleartext bits only for development-time validation and gate accounting. It provides no confidentiality and is never included in a deployment.

Implementation-level side channels We do not analyze microarchitectural side channels (cache timing and Spectre-class transient execution attacks) within the server's execution environment. Mitigations for these are deployment-level concerns shared by all cryptographic software.

4 Tailor Design

TAILOR is structured as a layered system that cleanly separates cryptographic primitives, integer arithmetic, neural operators, and the Python frontend. This section presents the overall architecture, the backend-agnostic interface that enables instantiation across multiple bitwise FHE schemes, the rationale for building arithmetic from bit-level circuits, and the encrypted integer and tensor abstractions on which higher-level operators are built. Operator-level constructions are deferred to Section 5.

4.1 Architectural Overview

Figure 2 illustrates TAILOR's four-layer architecture. The **backend layer** provides bitwise FHE primitives—encryption, decryption, and bootstrapped Boolean gates—through a uniform C++ template interface. The **arithmetic layer** composes these primitives into precision-parametric signed-integer circuits, exposed through an `IntCtxt` abstraction whose bit-width k is a runtime parameter. The **operator layer** implements neural-network operators—linear layers, convolution, and ReLU—over multi-dimensional `Tensor` objects of `IntCtxt` elements. Finally, the **frontend layer** exposes the operator layer to Python through `pybind11`, mirroring `torch.nn.Module` semantics so that PyTorch-trained models can be deployed with minimal modification.

A key design property of TAILOR is that *precision flows top-down as a software parameter, while cryptographic state remains anchored at the bottom.* When per-layer bit-widths are specified in the Python frontend, they propagate dynamically into the arithmetic

layer's circuit construction, but the backend layer's keys and security parameters remain unchanged.

4.2 Backend-Agnostic Bitwise Interface

TAILOR's arithmetic and operator layers are implemented against an abstract bitwise FHE interface rather than a concrete scheme. We realize this with a C++ template `Scheme<Backend>` that parameterizes all higher-level constructions on the underlying backend. The interface exposes six primitive operations:

```
template <typename Backend>
class Scheme {
  Ctxt Enc(bool m);
  bool Dec(const Ctxt& c);
  Ctxt AND (const Ctxt& a, const Ctxt& b);
  Ctxt OR (const Ctxt& a, const Ctxt& b);
  Ctxt XOR (const Ctxt& a, const Ctxt& b);
  Ctxt NOT (const Ctxt& a);
};
```

All bitwise FHE backends of interest—TFHE, FHEW, FINAL, and their variants—natively support these six operations. Bootstrapping noise management is encapsulated within each backend's gate implementation and is invisible to the upper layers. The `Scheme<Backend>` is deployed across two distinct bitwise FHE families and one plaintext reference:

- **TFHE** (LWE-based) [CGGI16] is the mainstream bitwise FHE construction, with bootstrapped gates around 7 ms. We use it as the default backend for performance-oriented evaluation.
- **FINAL** (NTRU-based) [BIP+22] employs matrix-NTRU ciphertexts with FHEW-style bootstrapping. We use it as a portability witness: identical TAILOR arithmetic and operator code compiles for both backends and produces bit-identical results.
- **Plain** (reference) implements the six primitives over cleartext bits and maintains an atomic count of bootstrapped-gate calls. It executes full networks in milliseconds, provides instant functional validation and exact gate accounting for the cost model in Section 5.4, and serves as a convenient debugging target. It offers no cryptographic protection and is excluded from the Section 3 threat model.

The instantiation effort is minimal. Porting from TFHE to FINAL required implementing only the six interface methods, without modifying any code in the arithmetic or operator layers. The plaintext backend comprises fewer than sixty lines of code.

4.3 Why Bit-Level Circuits over Programmable Bootstrapping

TFHE supports *programmable bootstrapping* (PBS), which evaluates a univariate lookup table (LUT) during noise refresh and enables the implementation of nonlinear functions in a single bootstrap. PBS-based frameworks such as Concrete-ML exploit this by encoding small integers as single ciphertexts and evaluating operators via table lookups. TAILOR, on the contrary, constructs operators from bit-level Boolean circuits for three critical reasons.

First, LUT size inherently scales with the polynomial degree. PBS evaluates a function $\phi : \mathbb{Z}_p \to \mathbb{Z}_p$ where p is bounded by half the underlying polynomial degree N. For $N = 1024$, the maximum input alphabet is $p \leq 512$, i.e., 9-bit integers. Beyond this limit, the operand must be decomposed into multiple ciphertexts and then recomposed—essentially reverting to a bit-sliced representation with additional overhead.

Second, bit-level circuits naturally support per-bit precision control. Because TAILOR's core contribution is runtime precision flexibility, a bit-sliced representation makes the precision-cost relationship transparent: each additional bit adds exactly one ciphertext and a predictable number of bootstrapped gates per operator. In contrast, PBS-based schemes tie precision to a discrete set of supported LUT sizes, each requiring its own bootstrapping key.

Third, bit-level circuits ensure backend portability. PBS is a TFHE-specific feature. NTRU-based backends, such as Final, provide bootstrapped Boolean gates but lack a PBS equivalent. Designing TAILOR around bit-level circuits preserves a uniform programming model across backend families and supports the multi-backend portability demonstrated in Section 7.3.2.

Ultimately, PBS-based and bit-sliced approaches are not strictly ordered: for individual small-arity operators, PBS can outperform the corresponding bit-level circuit. However, TAILOR's design prioritizes flexibility in precision and portability in backend selection over per-operator optimality—a trade-off we revisit in Section 7.4.

4.4 Encrypted Integer Representation

TAILOR represents an encrypted k-bit integer as an `IntCtxt` object, defined as a tuple $(\vec{c}, k, \tau)$, where $\vec{c} = (c_0, c_1, \ldots, c_{k-1})$ is a vector of bit ciphertexts, k is the bit-width, and $\tau \in \{\mathsf{signed}, \mathsf{unsigned}\}$ is the type tag. Bit c_0 is the least significant bit; for signed integers, c_{k-1} is the sign bit under two's-complement encoding. Encryption and decryption are performed bit-wise:

$$c_i \leftarrow \mathsf{Scheme.Enc}(x_i), \qquad x'_i \leftarrow \mathsf{Scheme.Dec}(c_i)$$

with reconstruction $x' = \sum_{i=0}^{k-1} x'_i \cdot 2^i$ for unsigned values, and $x' - 2^k$ when $\tau = \mathsf{signed}$ and $x'_{k-1} = 1$.

The bit-width k is set when an `IntCtxt` is constructed and may vary across operands within the same program. Arithmetic operators consume the k field of their inputs to determine the circuit shape and automatically zero- or sign-extend when operand widths differ. Moreover, k has *no relationship* to the backend's cryptographic parameters (N, q): changing k only changes the length of $\vec{c}$ and the number of dispatched gates.

A k-bit `IntCtxt` uses k times the memory of a single-bit ciphertext. This is the principal cost TAILOR pays for runtime precision flexibility. However, this representation is naturally parallel along the bit dimension: many arithmetic circuits (e.g., the ReLU mask) apply identical gates to each bit position, facilitating straightforward OpenMP parallelization.

4.5 Encrypted Tensor Abstraction

Neural-network workloads operate on multi-dimensional arrays. TAILOR provides a `Tensor<Backend>` abstraction that wraps a contiguous array of `IntCtxt` elements with shape metadata $(d_1, d_2, \ldots, d_n)$. Each element is encrypted independently, so tensor manipulations—slicing, padding, transposition, im2col reshaping—are implemented using standard C++ indexing logic and incur *no* homomorphic operations.

This stands in deliberate contrast to packed schemes (BGV, CKKS), where multiple plaintext values share a single ciphertext via SIMD slot encoding. Packed schemes amortize per-ciphertext cost across slots but require homomorphic rotations to access elements in different slot positions, complicating operations such as convolution and constraining tensor layouts to scheme-friendly shapes [CKK+23, YWH+24]. TAILOR's element-wise representation makes the opposite trade-off: higher memory usage in exchange for native indexing, arbitrary tensor shapes, and straightforward implementation of operators such as `Conv2D` via im2col.

5 Precision-Parametric Operators

This section presents the operator-level constructions that TAILOR composes on top of the `IntCtxt` and `Tensor` abstractions from Section 4. We describe signed-integer arithmetic, nonlinear operators, homomorphic neural layers, and an analytical cost model that maps operator parameters to gate counts.

5.1 Signed Integer Arithmetic

TAILOR's arithmetic operators follow standard digital-circuit constructions but are parameterized at the source level by the operand bit-widths and signedness flags carried by each `IntCtxt`, which are summarized in Table 2.

5.1.1 Addition

For two operands $a = (a_0, \ldots, a_{k_a-1})$ and $b = (b_0, \ldots, b_{k_b-1})$, TAILOR computes $s = a + b$ using a ripple-carry adder (RCA) of width $k = \max(k_a, k_b)$. Each bit position i produces a sum bit s_i and a carry-out c_{i+1}:

$$\begin{aligned} s_i &= a_i \oplus b_i \oplus c_i, \\ c_{i+1} &= (a_i \wedge b_i) \oplus (c_i \wedge (a_i \oplus b_i)), \end{aligned} \tag{1}$$

with $c_0 = \mathsf{Enc}(0)$. We use XOR rather than OR for the carry combination because the two terms are exclusive, saving one bootstrapped gate per bit position without altering the result.

When $k_a \neq k_b$, the shorter operand is virtually extended on the fly: unsigned operands are zero-padded with $\mathsf{Enc}(0)$, and signed operands are sign-extended by replicating the MSB. For $s = a + v$ with plaintext v, TAILOR inspects each bit of v at compile time. At positions where v's bit is zero, reduce the full adder to a wire pass-through, eliminating the corresponding AND and XOR gates. The optimization saves roughly $\bar{w}(v)/k$ fraction of gates, where $\bar{w}(v)$ is the plaintext Hamming weight.

5.1.2 Multiplication

Homomorphic integer multiplication is the dominant cost in neural-network workloads. A naive shift-and-add approach has $O(k)$ circuit depth, which compounds with the linear-layer dimension, resulting in impractical latency. TAILOR adopts a parallel multiplier structure inspired by hardware multipliers, reducing the multiplication depth to $O(\log k)$ at the cost of a quadratic gate count.

Given operands a and b of width k, TAILOR generates a $k \times k$ matrix of partial products, $pp_{i,j}$. For unsigned operands, $pp_{i,j} = a_i \wedge b_j$. For signed operands, we use the Baugh-Wooley transformation, which incorporates the sign correction into the partial-product matrix:

$$pp_{i,j} = \begin{cases} \neg(a_i \wedge b_j) & \text{if } (i = k-1) \oplus (j = k-1) \\ a_i \wedge b_j & \text{otherwise} \end{cases} \tag{2}$$

with two constant correction terms added to the appropriate columns. This avoids explicit sign conversion and keeps signed and unsigned code paths structurally identical.

The partial-product matrix is reduced using a tree of carry-save adders (CSAs). Each CSA takes three partial-product vectors u, v, and w at the same bit-weight positions and products at the same bit weight and produces a sum vector S and a carry vector C:

$$\begin{aligned} S &= u \oplus v \oplus w, \\ C &= (u \wedge v) \oplus (w \wedge (u \oplus v)). \end{aligned} \tag{3}$$

The carry vector is shifted left by one position before the next reduction round. Each CSA layer is independent across bit positions, the layer can be naturally parallelized using OpenMP. The reduction continues until two vectors remain, which are then summed with the ripple-carry adder.

For $p = a \cdot v$ with plaintext v, TAILOR avoids generating a full $k \times k$ partial-product matrix. Special cases $v \in \{-1, 0, 1\}$ are handled by negation, zero-encryption, or a copy. For other v, we decompose $|v|$ into binary and emit one shifted copy of a for each set bit; the sign of v is applied via a final negation. The shifted copies are accumulated using a balanced binary tree of homomorphic adders, yielding a total depth of $O(\log w(v))$ where $w(v)$ is the Hamming weight of v. Products are materialized at their exact width $k + \text{bits}(v)$—where $\text{bits}(v)$ denotes the two's-complement width of v—rather than at a fixed $2k$, so no gate is ever spent on high-order bits that are provably zero.

5.2 Nonlinear Operators

A central advantage of bit-level FHE is that nonlinear functions can be evaluated exactly using Boolean circuits, without polynomial approximation. TAILOR provides three nonlinear primitives that compose to form common neural activations.

5.2.1 Absolute value

For a signed integer x with sign bit $s = x_{k-1}$, we compute $|x|$ using conditional negation in two's complement:

$$|x| = (x \oplus s) + s, \tag{4}$$

where s is broadcast to a vector of length k. When $s = 0$, the expression reduces to x; when $s = 1$, it computes the bitwise NOT of x plus one, which is precisely the two's-complement negation. For unsigned operands, $|x|$ is the identity.

5.2.2 Min and max

We implement comparisons by inspecting the sign of the difference $d = a - b$. To prevent overflow from affecting the comparison, both operands are sign-extended to $k + 1$ bits before subtraction. The MSB of d serves as the selector bit s:

$$s = (a - b)_k. \tag{5}$$

The min and max are then selected by a homomorphic multiplexer built from XOR and AND gates:

$$\mathsf{Min}(a, b) = b \oplus (s \wedge (a \oplus b)), \tag{6}$$

$$\mathsf{Max}(a, b) = b \oplus (\neg s \wedge (a \oplus b)). \tag{7}$$

When $s = 1$, the expression for Min reduces to a; otherwise, it reduces to b, with the dual behavior for Max. This formulation requires roughly twice as many gates as an addition, substantially fewer than the arithmetic identity $\max(a, b) = (a + b + |a - b|)/2$ which involves an extra absolute value and an integer halving step.

5.3 Homomorphic Neural Layers

The integer and nonlinear primitives combine to form the three neural-network layers we evaluate in Section 7: linear (fully connected), 2D convolution, and ReLU activation.

5.3.1 Linear layer

Given an encrypted input $x \in \mathbb{Z}^{N_{\text{in}}}$, a plaintext weight matrix $W \in \mathbb{Z}^{N_{\text{out}} \times N_{\text{in}}}$, and bias $b \in \mathbb{Z}^{N_{\text{out}}}$, the linear layer computes $y = Wx + b$. Because naive sequential accumulation yields a prohibitive $O(N_{\text{in}})$ additive depth, Tailor applies three system-level optimizations: (1) *Sparsity awareness:* We skip all partial products where $W_{m,k} = 0$, directly exploiting the exact zeros prevalent in post-training quantization to eliminate scalar multiplications and additions. (2) *Tree-based reduction:* Non-zero products are accumulated via a dynamically constructed, balanced binary tree of homomorphic adders, reducing the critical additive depth per neuron from $O(N_{\text{in}})$ to $O(\log N_{\text{in}})$. Tree adders operate at progressively increasing widths—each level widens partial sums by exactly one bit—so no addition ever exceeds what is provably necessary. (3) *Two-level parallelism:* Because the reduction trees for each output neuron are independent, we parallelize the N_{out} computations across CPU cores using OpenMP. When a layer has fewer output neurons than hardware threads, the scheduler switches to a second mode: all scalar multiplications are flattened into a single global work queue, and each tree level is then processed synchronously across neurons, thereby recovering intra-neuron parallelism on narrow layers, such as a 10-way classifier head.

Accumulators are provisioned analytically rather than heuristically: given input width k_{in}, weight width k_w, and maximum non-zero fan-in n_{nz} (computed per layer at graph-construction time), the accumulator width $k_{\text{acc}} = k_{\text{in}} + k_w + \lceil \log_2 n_{\text{nz}} \rceil$ is the minimum width that provably prevents overflow. Pruning therefore reduces not only the number of homomorphic operations but also the bit width at which the surviving operations execute.

5.3.2 Convolution via im2col

Two-dimensional convolution is implemented by reducing it to general matrix multiplication (GEMM). Given an encrypted input tensor $A \in \mathbb{Z}^{C \times H \times W}$ and a plaintext weight tensor of shape (M, C, k_h, k_w), the convolution proceeds in three stages: (i) the input is transformed into a matrix $A_{\text{col}} \in \mathbb{Z}^{N \times K}$ via the standard *im2col* reshaping, where N is the number of output spatial positions and $K = C \cdot k_h \cdot k_w$; (ii) the weights are reshaped into a matrix $W_{\text{mat}} \in \mathbb{Z}^{K \times M}$; (iii) the output is computed by applying the linear-layer routine from Section 5.3.1 to each output channel.

Because Tailor's `Tensor` stores each pixel as an independent `IntCtxt`, the im2col transformation is implemented purely through index manipulation in C++ without any homomorphic operations. Padding is handled during indexing: positions outside the input boundary are filled with $\mathsf{Enc}(0)$, producing exact zero-padding behavior without altering the input tensor. This contrasts with packed-scheme implementations, where rearranging slot positions for convolution requires homomorphic rotations whose cost increases with kernel size. Convolutional layers share the same per-layer precision signature as linear layers—weight width, requantization shift, output width, and an optional fused activation—so the conclusion in Section 5.3.4 applies uniformly to both convolutional and linear layers.

5.3.3 ReLU via bit masking

Unlike arithmetic frameworks that rely on inaccurate polynomial approximations or hybrid protocols that compromise end-to-end encryption, Tailor achieves an exact ReLU via a constant-depth bit-masking circuit. In two's-complement representation, an integer x is non-negative if and only if its sign bit $x_{k-1} = 0$. We therefore extract a mask bit $m = \neg x_{k-1}$ and apply a bitwise AND across all bit positions: $y_i = x_i \wedge m$ for all $i \in \{0, \ldots, k-1\}$. Consequently, non-negative inputs are preserved exactly ($m = 1$) while negative inputs are completely zeroed ($m = 0$). This design requires only a single bootstrapped AND gate per output bit—an $O(1)$ depth entirely independent of k. In complete networks this

Table 2: Bootstrapped-gate counts for TAILOR's operators.

Operator	Bootstraps	Depth
Add(a, b)	$3k$	$O(k)$
Mul(a, b)	$O(k^2)$	$O(\log k)$
RMul(a, v)	$O(k \cdot w(v))$	$O(\log w(v))$
Abs(a)	$2k$	$O(1)$
Min(a, b)	$5k$	$O(k)$
Max(a, b)	$5k$	$O(k)$
ReLU(a)	k	$O(1)$
ReqReLU(a, s)	$W + 4(W-s+1)$	$O(W-s)$
Linear	$N_{\text{out}}\left[\,\overline{G_{\text{RMul}}} + 3k(\bar{n}_{\text{in}} - 1)\right]$	$O(\log k + \log N_{\text{in}})$

mask rarely runs standalone: it is embedded in the fused requantization post-processing described next.

5.3.4 Fused requantization–ReLU post-processing

Quantized inference inserts a requantization step after every linear or convolutional layer: the wide accumulator must be rescaled by a power-of-two factor and clamped to the next layer's activation range $[0, 2^k-1]$. TAILOR fuses this step with activation into a single post-processing operator, ReqReLU, which maps a W-bit signed accumulator directly to a k-bit *unsigned* activation. The circuit proceeds in three stages: (i) the bit-masking ReLU sets negative accumulators to zero; (ii) a round-to-nearest power-of-two shift discards the s least-significant bits while homomorphically adding back the bit at position $s-1$, thereby matching integer round-half-up semantics; and (iii) a saturation stage ORs the surviving bits above position k into an overflow flag, which is then ORed into every output bit, clamping any out-of-range value to the maximum representable activation, 2^k-1.

The post-processing costs $W + 4(W-s+1)$ bootstrapped gates and yields three benefits. First, the standalone activation pass is removed from the execution graph. Second, outputs are emitted *unsigned*, saving one ciphertext per activation, as required by the Section 6.1 quantization pipeline. Third, the saturating clamp reproduces the quantization-aware training graph's activation range bit-for-bit, preserving exact correspondence between encrypted and plaintext integer inference. When a layer's ReLU flag is unset (e.g., the final logit layer), the post-processing reduces to the rounding shift alone at a caller-selected output width.

5.4 Cost Model

The runtime of a homomorphic neural network in TAILOR is determined almost entirely by the number of bootstrapped Boolean gates dispatched to the backend. Encryption, decryption, and NOT gates execute in microseconds, whereas bootstrapped AND/OR/XOR each require several milliseconds; we quantify this gap empirically in Section 7.2. The total inference latency is therefore well approximated by

$$T \approx G_{\text{boot}} \cdot t_{\text{boot}}/P, \tag{8}$$

where G_{boot} is the total bootstrapped gate count, t_{boot} is the per-gate bootstrapping latency of the chosen backend, and P is the effective OpenMP parallel speedup.

Table 2 reports G_{boot} for each operator as a function of bit-width k. The Linear-layer count comprises two components: $N_{\text{out}} \cdot \overline{G_{\text{RMul}}}$ bootstraps for scalar multiplications between encrypted activations and plaintext weights, plus $3k \cdot N_{\text{out}}(\bar{n}_{\text{in}} - 1)$ bootstraps for tree-based accumulation of partial products. Here, $\bar{n}_{\text{in}}$ denotes the average number

of non-zero weights per output neuron, and $\overline{G_{\mathrm{RMul}}}$ averages the RMul cost over those non-zero positions. The tree reduction does not change the total number of additions (still $\bar{n}_{\mathrm{in}} - 1$ per output neuron) but reduces the circuit depth from $O(N_{\mathrm{in}})$ to $O(\log N_{\mathrm{in}})$, which is the latency-relevant quantity under sufficient parallelism. For ReqReLU, W denotes the accumulator width and s the requantization shift. With the progressive-width reduction in Section 5.3.1, level-ℓ tree adders operate on $(k+\ell)$-bit operands, so evaluating the accumulation term at the full accumulator width yields an upper bound.

6 FHE-Aware Quantization and Deployment

Tailor's runtime precision flexibility is useful only when trained models produces weights and activations that match its bit-sliced integer representation. Standard quantization-aware training (QAT) toolchains, however, target hardware accelerators whose numeric conventions differ from FHE in subtle but consequential ways. This section describes the design choices we make on top of Brevitas [FPF25] to align QAT with Tailor's execution model, the sparsity-aware training schedule that leverages Tailor's zero-skipping optimization, and the end-to-end deployment stack (Section 6.3).

6.1 Aligning Brevitas with Tailor

Standard quantization-aware training (QAT) toolchains like Brevitas simulate fixed-point hardware but ultimately export floating-point scale factors optimized for conventional accelerators. This mismatches Tailor's execution model, which operates exclusively on encrypted two's-complement integers and lacks an efficient representation for arbitrary real-valued scale factors. To resolve this representational gap, we impose three FHE-aware design constraints on the Brevitas pipeline:

1. *Power-of-two scaling.* We adopt `FixedPoint` quantizer in Brevitas to restrict all scale factors to powers of two. Under Tailor's bit-sliced representation, this constraint reduces the application of scale factors from an expensive homomorphic multiplication to a completely free logical re-indexing of ciphertexts.

2. *Unsigned post-ReLU activations.* Because ReLU outputs are strictly non-negative, configuring the post-activation quantizer as unsigned saves exactly one ciphertext per activation. Tailor's downstream operators natively process these unsigned inputs against signed plaintext weights without any conversion overhead. The fused version in Section 5.3.4 implements this constraint exactly in the encrypted domain: its saturating clamp reproduces the quantizer's $[0, 2^k - 1]$ range bit-for-bit.

3. *Bias-free linear layers.* In standard QAT, biases are stored at high precision (e.g., 32-bit) to prevent accumulator overflow. We completely omit bias terms during training, eliminating the need to materialize wide-precision encrypted bias tensors and execute expensive homomorphic additions, with negligible accuracy loss on our evaluated workloads.

6.2 Sparsity-Aware Training Schedule

Tailor's linear layer skips plaintext-zero weights entirely, eliminating both the scalar multiplication and the corresponding tree-reduction addend. In a bit-wise FHE setting, this is more consequential than on hardware accelerators: a single skipped weight saves on the order of $k \cdot w(W_{m,j})$ bootstrapped gates, each costing serveral milliseconds at $\lambda = 128$. Sparsity is therefore not merely a memory optimization but a first-order latency optimization.

Table 3: Progressive sparsity schedule applied during QAT.

Epochs	FC1 sparsity	FC2 sparsity
0–9	0%	0%
10–12	30%	20%
13–15	50%	30%
16–18	70%	40%
19–21	80%	50%
22–29	90%	50%

We exploit this with a *progressive unstructured pruning* schedule, shown in Table 3, during QAT. Starting at epoch 10, we apply L1-norm unstructured pruning to the two linear layers with target sparsities that increase every three epochs, reaching final sparsities of 90% for FC1 (784→256) and 50% for FC2 (256→10) by epoch 22. The schedule is asymmetric by design: FC1 contains ∼200K weights and dominates inference latency, making it the high-value pruning target, while FC2 contains only 2.5K weights and is more sensitive to aggressive pruning.

6.3 PyTorch-to-Tailor Deployment

Tailor exposes its operator layer to Python through `pybind11` bindings. A trained Brevitas model is deployed in three steps: (i) export weights as integer tensors using the FHE-aware quantizer; (ii) instantiate a Tailor engine and bind the integer tensors to homomorphic layers; (iii) encrypt the input and invoke the engine.

Requantization shifts are derived, not hand-tuned: at export time, the pipeline reads the power-of-two scale factors attached to each Brevitas quantizer and computes each shift arithmetically—e.g., $s_1 = \log_2 s_{\text{act}} - \log_2 s_{\text{in}} - \log_2 s_{w_1}$ for the hidden layer—so the encrypted graph and the integer reference provably implement the same fixed-point program. The final layer's logit width is likewise automatically set based on the observed dynamic range.

Listing 1 illustrates the deployment of a mixed-precision MLP. Each layer carries its full precision signature—weight width `wt_nbits`, requantization `shift`, output width `out_nbits`, and a ReLU flag that selects the fused post-processing—so switching precision configurations amounts to changing per-layer keyword arguments:

Listing 1: End-to-end deployment with Tailor.

```
scheme = TFHEScheme()     # or NTRUScheme(),
engine = Engine(scheme)   #    PlainScheme()

data = engine.Input()
fc1  = engine.Linear(data, fc1_w, fc1_b,
         wt_nbits=k_w1, shift=s1,
         out_nbits=k_act, relu=True)
out  = engine.Linear(fc1, fc2_w, fc2_b,
         wt_nbits=k_w2, shift=s2,
         out_nbits=k_out, relu=False)

ct  = scheme.encrypt(image, nbits=k_in)
res = engine.run({data: ct})
```

The deployment pipeline strictly separates graph construction from homomorphic execution. Developers select a backend by instantiating a target scheme (e.g., `TFHEScheme`, `NTRUScheme`, or `PlainScheme`). With the `Scheme<Backend>` abstraction, all downstream model definitions remain identical; switching cryptographic backends requires changing

Table 4: Per-gate latency.

Gate	TFHE-rs (ms)	FINAL (ms)	Ratio
AND	7	59	8.4×
NAND	7	59	8.4×
OR	7	59	8.4×
XOR	24	58	2.4×

only a single line of Python. Furthermore, invoking operators like `engine.Linear` does not trigger immediate computation. Instead, these calls intuitively construct a directed acyclic graph (DAG) in C++ memory, binding the precomputed plaintext weights. The final `engine.run` invocation traverses this DAG to dispatch workloads to the multi-threaded backend. This deferred-execution architecture keeps the Python frontend lightweight while strictly confining all hot-path cryptography to highly optimized C++ or Rust.

7 Evaluation

We evaluate Tailor along five axes: (i) per-gate microbenchmarks; (ii) precision-dependent integer arithmetic and cost-model validation; (iii) end-to-end MLP inference on MNIST; (iv) reconfiguration cost against representative FHE inference frameworks; and (v) generalization of the QAT pipeline to CIFAR-10.

7.1 Experimental Setup

All experiments run on a single AWS `c8a.48xlarge` instance (AMD EPYC, 192 vCPUs, 384 GB RAM) using Ubuntu 24.04 and GCC 13.3.0. Models are quantized and trained via PyTorch 2.10.0 and Brevitas 0.12.1. For homomorphic execution, we evaluate two bitwise backends instantiated at a standard $\lambda = 128$-bit security level: **TFHE-rs** [Zam22b] (the reference Rust implementation) and a thread-safe adaptation of **FINAL** [BIP+22]. A third instantiation, the plaintext reference backend **Plain**, executes the same operator graph over cleartext bits to count bootstrapped gates. We use it for functional validation, exact gate accounting, and as a sanity baseline in the end-to-end comparison in Section 7.5.

To rigorously isolate performance bottlenecks, we enforce a strict threading convention throughout our evaluation. Boolean-gate and integer-operator microbenchmarks execute *single-threaded* to establish intrinsic hardware costs without scheduling overhead, validating the analytical model of Section 5.4. Conversely, neural-network operators and end-to-end inference simulate realistic deployments by utilizing all 192 threads via OpenMP, effectively parallelizing the outer tensor dimensions while keeping inner arithmetic sequential.

7.2 Backend Microbenchmarks

We first evaluate the per-gate latency of Tailor's supported backends. Because all higher-level constructions decompose into these Boolean primitives, these measurements ground the analytical cost model and contextualize our end-to-end results.

Table 4 details these microbenchmarks, revealing three critical performance dynamics that inform the rest of our evaluation:

(1) *Bootstrapping dominates execution:* While encryption, decryption, and unary NOT operations complete in under 1 ms, bootstrapped binary gates require several milliseconds. This validates our cost-model assumption that end-to-end latency is overwhelmingly bounded by the bootstrapped-gate count.

(2) *TFHE-rs provides a substantial baseline advantage:* TFHE-rs evaluates AND/OR/-NAND in just 7 ms compared to Final's 59 ms. This 8.4× speedup reflects the maturity of

Table 5: Latency of TAILOR's integer operators on TFHE-rs under worst-case operands, in milliseconds.

Operator	$k = 2$	$k = 4$	$k = 8$	**Scaling**
Add	74	143	272	$O(k)$
Abs	90	171	314	$O(k)$
Min	193	320	566	$O(k)$
Max	194	318	571	$O(k)$
RMul ($v = -1$)	58	113	212	$O(k)$
Mul	292	1091	4440	$O(k^2)$

the optimized Rust LWE implementation. As a result, we adopt TFHE-rs as the default performance baseline, retaining Final strictly to witness cross-scheme portability.

(3) *TFHE-rs exhibits asymmetric gate costs:* Unlike Final, which maintains a uniform 58–59 ms cost across all gates, TFHE-rs evaluates XOR at 24 ms—roughly 3.4× slower than AND/OR. This occurs because TFHE-rs optimizes AND/OR/NAND via a fused bootstrapping path, but relegates XOR to a more general programmable bootstrap. This asymmetry directly impacts TAILOR's higher-level arithmetic: operators with XOR on their critical path incur a weighted latency penalty rather than a uniform per-gate cost.

7.3 Integer Arithmetic

Building on the gate-level baseline, we evaluate TAILOR's precision-parametric integer operators. To capture adversarial or deployment worst-case latency, we benchmark signed, single-threaded operators using all-ones operands ($a = -1, b = -1, v = -1$) over 10^3 iterations, intentionally bypassing data-dependent fast paths.

As shown in Table 5, empirical latencies closely match the analytical cost model (Table 2). Operators sharing the ripple-carry backbone—Add, Abs, Min, and Max—scale linearly. For instance, doubling k from 4 to 8 incurs merely a 1.77×–1.90× penalty. The Abs operator (314 ms at $k = 8$) leverages the conditional-negation circuit to perform nearly as fast as a standard Add (272 ms), validating the omission of complex sign-extraction logic. Similarly, multiplexer-based Min/Max (566/571 ms) requires roughly twice the Add latency, outperforming traditional arithmetic identities that necessitate multiple full-precision operations.

Moreover, the Wallace-tree Mul exhibits expected $O(k^2)$ scaling, jumping 15× (292 ms to 4440 ms) as operands widen from 2 to 8 bits. Plaintext scalar multiplication (RMul) displays extreme data dependency: computing $v = -1$ resolves to a linear-time homomorphic negation (212 ms at $k = 8$), whereas an all-ones plaintext ($v = 2^k - 1$) forces near-quadratic evaluation (3588 ms). TAILOR directly exploits this dynamic range to accelerate linear layers.

7.3.1 GEMM Scaling and Cost-Model Validation

To evaluate realistic neural workloads, we benchmark a signed 32×32 general matrix multiplication (GEMM)—the core computational kernel for linear and convolutional layers. We utilize OpenMP to parallelize across the 1024 output elements, fully saturating the 192 available hardware threads.

Table 6 demonstrates that under full parallelism, GEMM latency adheres tightly to the $O(k^2)$ Mul prediction. Expanding k from 4 to 8 scales latency by 4.20× (within 5% of theoretical). Scaling from $k = 2$ to $k = 4$ yields 3.68×, slightly below quadratic due to the linear-time Add tree-reduction occupying a larger relative proportion at extreme low precisions.

Table 6: Signed 32×32 GEMM latency on TFHE-rs.

k	**Latency (s)**	**vs.** $k = 2$
2	53.6	1.0×
3	110.6	2.1×
4	197.3	3.7×
5	308.3	5.8×
6	462.7	8.6×
7	620.9	11.6×
8	828.3	15.5×

Table 7: Signed integer-operator latency (ms).

Operator ($k = 8$)	**TFHE-rs**	**FINAL**	**Ratio**
Add	272	2,333	8.6×
Mul	4,440	39,849	9.0×
Abs	314	2,799	8.9×
Min	566	5,071	9.0×
Max	571	5,079	8.9×
RMul ($v = -1$)	212	1,866	8.8×

7.3.2 Backend Portability Witness

We validate Tailor's backend-agnostic design by re-executing all $k = 8$ microbenchmarks on the FINAL backend. As shown in Table 7, both backends yield bit-identical outputs. FINAL operates 8.6×–9.0× slower than TFHE-rs, cleanly mirroring the 8.4× gate-level latency gap observed in Table 4 (with minor variance driven by XOR proportions). This consistency demonstrates that Tailor's arithmetic layer faithfully exposes the underlying backend cost without introducing scheme-specific overhead.

7.4 End-to-End MLP Inference on MNIST

We evaluate Tailor end-to-end using a two-layer MLP ($784 \to 256 \to 10$) for MNIST classification. The model undergoes our FHE-aware QAT pipeline and progressive pruning (Table 3), achieving final sparsities of 90% (FC1) and 50% (FC2). Table 8 reports per-layer latency, total latency, and test-set accuracy across seven precision configurations, evaluated on 10 images using TFHE-rs with 192 threads. The FC1 column includes the fused requantize–ReLU post-processing, so no standalone activation pass remains.

Table 8 yields two critical system insights. First, activation handling is now effectively free: the fused post-processing folds ReLU, round-to-nearest rescaling, and saturation into the tail of each linear layer, so no standalone activation pass appears in the execution graph. Its cost—one masking AND per accumulator bit plus a single rounding addition per neuron—is absorbed into the reported FC1 latency. Second, end-to-end latency shows highly compressed scaling: a 4-fold precision increase ($k = 2$ to $k = 8$) yields only a 5.8× latency penalty (109.8 s to 632.4 s), well below the theoretical 16× scaling of pure $O(k^2)$ multiplication. This compression highlights the efficacy of our sparsity-aware tree reduction: layer latency is ultimately bottlenecked by the additive accumulation step, whose cost grows with the number of non-zero weights and near-linearly with the accumulator width rather than quadratically with per-multiplication bit-width.

The accuracy–latency Pareto frontier comprises only low- and mixed-precision configurations: $(2, 2, 2, 2)$ at 109.8 s/94.44%, $(4, 2, 4, 4)$ at 158.2 s/95.80%, and $(4, 4, 4, 2)$ at 222.0 s/96.44%. The $(4, 4, 4, 2)$ setup strictly Pareto-dominates both uniform baselines: it achieves the highest accuracy of all seven configurations and runs faster than uniform 4-bit

Table 8: End-to-end MLP inference latency on TFHE-rs.

Config	Latency (s)			Acc.
$(k_{\text{in}}, k_{w1}, k_{\text{act}}, k_{w2})$	FC1+ReLU	FC2	Total	(%)
$(2, 2, 2, 2)$	70.4	39.4	109.8	94.44
$(4, 4, 4, 4)$	171.7	64.4	236.1	96.17
$(8, 8, 8, 8)$	510.6	121.7	632.4	96.14
$(2, 4, 4, 4)$	138.8	67.6	206.4	94.84
$(4, 2, 4, 4)$	95.3	62.9	158.2	95.80
$(4, 4, 2, 4)$	170.8	55.3	226.1	95.65
$(4, 4, 4, 2)$	175.0	46.9	222.0	96.44

Table 9: End-to-end MLP latency across backends. Ratio is FINAL over TFHE-rs.

Config	Plain (ms)	TFHE-rs (s)	FINAL (s)	Ratio
$(2, 2, 2, 2)$	7.5	109.8	721.3	6.6×
$(4, 4, 4, 4)$	16.0	236.1	1,493.6	6.3×
$(8, 8, 8, 8)$	44.6	632.4	3,827.2	6.1×
$(2, 4, 4, 4)$	98.0	206.4	1,312.6	6.4×
$(4, 2, 4, 4)$	96.0	158.2	1,020.3	6.5×
$(4, 4, 2, 4)$	15.4	226.1	1,398.4	6.2×
$(4, 4, 4, 2)$	15.9	222.0	1,428.1	6.4×

and 2.8× faster than uniform 8-bit. The frontier's shape is structurally unpredictable from uniform runs alone: halving the FC1 weight width $(4, 2, 4, 4)$ cuts FC1 latency by 1.8× at a modest 0.37 pp accuracy cost, whereas halving the activation width $(4, 4, 2, 4)$ saves almost nothing and is itself dominated. Blindly maximizing precision remains actively detrimental: a uniform 8-bit configuration is the slowest yet fails to beat even uniform 4-bit accuracy. By exposing bit-width as a software parameter, Tailor empowers deployers to exploit these nontrivial Pareto optima without introducing any cryptographic or compilation overhead.

7.5 End-to-End Inference across Backends

The portability witness in Section 7.3.2 established operator-level equivalence; we now extend this to support encrypted inference. Table 9 reports the total MLP latency across all seven precision configurations for the three `Scheme<Backend>` instantiations. All backends produce bit-identical predictions on every configuration.

Two observations follow. First, FINAL runs 6.1×–6.6× slower than TFHE-rs end-to-end, below the 8.6×–9.0× operator-level gap in Table 7. This narrowing reflects the end-to-end gate mix and the non-bootstrapping phases of full inference, which reduce the per-gate advantage. As Section 7.2 notes, the exact ratio depends on each workload's XOR proportion. Second, the plaintext backend executes the same graphs in 7–98 ms—four orders of magnitude faster—making it practical to validate every candidate precision configuration and determine its exact bootstrapped-gate count before committing to encrypted execution.

7.6 Reconfiguration Cost across Frameworks

To quantitatively justify Tailor's decoupled architecture, we benchmark its deployment overhead against three representative frameworks: SEAL BFV, SEAL CKKS, and Concrete-

Table 10: Cryptographic state overhead.†: analytically estimated.

Framework	Configuration	Setup (s)	Keys (MB)
SEAL BFV	$k = 2$	0.14	33.8
	$k = 4$	0.13	33.8
	$k = 8$	0.13	33.8
SEAL CKKS	depth 3	0.21	44.4
	depth 7	1.28	396.4
	depth 16	10.98	414.0†
	depth 19	15.58	514.5†
Concrete-ML	$k = 2$	0.38	0.1
	$k = 4$	1.19	0.2
	$k = 8$	*compile failure*	
Tailor-TFHE	**any precision**	**0.14**	**124.4**
	per-switch	**0**	**0**

ML. We measure the wall-clock time and serialized key size required to prepare the cryptographic state for our MNIST MLP. For baseline frameworks, this setup cost must be paid per precision configuration (re-keying or recompilation). For Tailor, this is a strict one-time cost. All measurements are averaged over 3–5 runs on our standard `c8a.48xlarge` instance.

Table 10 exposes three distinct structural limitations in existing frameworks. First, while SEAL BFV imposes a modest 0.13 s keygen latency, it demands a fresh 33.8 MB key set for every precision change. Exploring the seven precision configurations from Table 8 would require repeatedly redistributing 236.6 MB of key material. Second, SEAL CKKS key sizes scale polynomially with the circuit's multiplicative depth. A shallow depth-3 configuration requires 44.4 MB, but scaling to depth 16 explodes the in-memory key footprint to 3.6 GB—fundamentally breaking the 2 GB single-message limit of Protocol Buffers. This complicates client–server architectures that rely on atomic key delivery.

Third, Concrete-ML suffers from a rigid precision ceiling dictated by its PBS-centric design. Because programmable bootstrapping evaluates functions via univariate lookup tables bounded by the polynomial degree, Concrete-ML imposes a strict 16-bit limit on intermediate accumulations. Consequently, attempting to compile our MLP at $k = 8$ fails because the FC1 accumulator requires 20 bits ($\lceil \log_2(784 \times 255 \times 127) \rceil$). This forces developers to restructure models or artificially restrict precision.

Tailor cleanly bypasses all three failure modes. Its bit-sliced architecture guarantees a fixed, depth-independent 124.4 MB evaluation key. This single key allocation serializes cleanly, requires only a one-time network transmission, and supports arbitrary software-level precision sweeps with exactly zero reconfiguration overhead.

7.7 Generalization to CIFAR-10

To validate that Tailor's FHE-aware QAT pipeline generalizes to more complex vision tasks, we train a quantized VGG-style CNN on CIFAR-10 (three 3×3 conv blocks, two FC layers, progressive pruning). We denote configurations as $(k_{\text{in}}, k_{c_1}, k_{c_2}, k_{c_3}, k_{\text{fc}})$, where k_{c_i} and k_{fc} are the bit-widths for the conv blocks and fully-connected layers, respectively.

Table 11 highlights three key dynamics of mixed-precision quantization. First, convolutional precision dictates overall accuracy: holding FC at 4-bit and raising conv to 8-bit yields a +0.90% gain (configuration $(8, 8, 8, 8, 4)$), whereas the reverse yields no improvement. This underscores that early quantization noise in feature extraction cannot be recovered downstream. Second, mixed precision can outperform uniform baselines.

Table 11: CIFAR-10 classification accuracy.

Configuration $(k_{\mathbf{in}}, k_{c_1}, k_{c_2}, k_{c_3}, k_{\mathbf{fc}})$	**Accuracy (%)**
Uniform precision	
$(8, 8, 8, 8, 8)$	86.44
$(8, 4, 4, 4, 4)$	85.71
Mixed precision variants	
$(8, 8, 8, 8, 4)$	**86.61**
$(8, 4, 4, 4, 8)$	85.60
$(8, 4, 4, 8, 4)$	85.55
$(4, 4, 4, 8, 4)$	84.90

The $(8, 8, 8, 8, 4)$ configuration achieves the highest accuracy (86.61%) while halving the bit-width of FC1's $\approx$ 524,000 weights, which under Tailor's bit-sliced execution directly translates to massive gate-count savings. Third, unlike MNIST, CIFAR-10's natural images penalize aggressive input quantization; dropping k_{in} from 8 to 4 incurs a 0.65% penalty. Tailor allows deployers to navigate these trade-offs transparently via software parameters.

8 Discussion and Conclusion

We presented Tailor, an end-to-end framework for privacy-preserving neural inference that fundamentally severs the traditional binding between arithmetic precision and cryptographic state. By representing integers as bit-sliced ciphertexts and constructing precision-parametric Boolean circuits, Tailor elevates bit-width to a purely software-defined deployment knob. Our evaluation demonstrates that this architecture unlocks mixed-precision configurations that simultaneously improve both accuracy and inference latency over uniform-precision baselines.

Despite these architectural contributions, several avenues remain for extending Tailor's capabilities. One direction is to push parallelism beyond individual layers: the two-level GEMM scheduler in Section 5.3.1 already partitions scalar multiplications and tree-reduction levels across threads within a layer, and overlapping the execution of consecutive layers in a pipelined fashion would build naturally on this foundation. Furthermore, while exact integer arithmetic covers a broad spectrum of PPML workloads, extending Tailor's Boolean circuits to support **block-floating-point formats** would expand its applicability to dynamic sequence models and attention mechanisms.

Essentially, the `Scheme<Backend>` abstraction natively positions Tailor for **hardware acceleration**. Because inference latency is strictly bottlenecked by Boolean gate evaluations, integrating GPU- or FPGA-accelerated backends would yield order-of-magnitude speedups with zero modifications to the frontend or operator logic. Finally, while extending to convolutional architectures is mechanically straightforward, exactly evaluating transcendental functions (e.g., exponentials for LayerNorm or Softmax) via bit-level circuits remains a substantial open challenge.

Ultimately, Tailor establishes runtime-configurable precision as a practical, deployable primitive for FHE-based machine learning. By prioritizing strict backend-agnosticism and exact bit-level correctness, it provides a robust architectural foundation for next-generation privacy-preserving systems that demand deployment flexibility, adaptive performance, and uncompromising security.

References

[AAB+23] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkowich, Dov Murik, Hayim Shaul, and Omri Soceanu. HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. In *Proceedings on Privacy Enhancing Technologies (PoPETs)*, 2023.

[BCCW19] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data. arXiv preprint arXiv:1908.04172, aug 2019.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325. Association for Computing Machinery, 2012.

[BIP+22] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. Final: Faster fhe instantiated with ntru and lwe. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology – ASIACRYPT 2022*, pages 188–215. Springer Nature Switzerland, 2022.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, pages 868–886. Springer Berlin Heidelberg, 2012.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. Cryptology ePrint Archive, Paper 2016/870, 2016.

[CGGI18] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. Cryptology ePrint Archive, Paper 2018/421, 2018.

[CKK+23] Jung Hee Cheon, Minsik Kang, Taeseong Kim, Junyoung Jung, and Yongdong Yeo. Batch inference on deep convolutional neural networks with fully homomorphic encryption using channel-by-channel convolutions. Cryptology ePrint Archive, Paper 2023/632, 2023.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437. Springer International Publishing, 2017.

[DGBL+16] Nathan Dowlin, Ran Gilad-Bachrach, K. Laine, Kristin E. Lauter, M. Naehrig, and J. Wernsing. Cryptonets: applying neural networks to encrypted data with high throughput and accuracy. In *ICML 2016*, 2016.

[DKS+20] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. Eva: an encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '20, pages 546–561. ACM, June 2020.

[DM14] Léo Ducas and Daniele Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. Cryptology ePrint Archive, Paper 2014/816, 2014. https://eprint.iacr.org/2014/816.

[DSC+19] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. Chet: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 142–156. Association for Computing Machinery, 2019.

[FPF25] Giuseppe Franco, Alessandro Pappalardo, and Nicholas J Fraser. Xilinx/brevitas, 2025.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012. https://eprint.iacr.org/2012/144.

[JMPP24] Robin Jadoul, Axel Mertens, Jeongeun Park, and Hilder V. L. Pereira. Ntru-based fhe for larger key and message space. Cryptology ePrint Archive, Paper 2024/634, 2024. https://eprint.iacr.org/2024/634.

[KHSS22] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. Scaling up trustless dnn inference with zero-knowledge proofs. arXiv preprint arXiv:2210.08674, 2022.

[VKH23] Alexander Viand, Christian Knabenhans, and Anwar Hithnawi. Verifiable fully homomorphic encryption. arXiv preprint arXiv:2301.07041, 2023.

[WZB+23] Qifan Wang, Lei Zhou, Jianli Bai, Yun Sing Koh, Shujie Cui, and Giovanni Russello. Ht2ml: An efficient hybrid framework for privacy-preserving machine learning using he and tee. *Computers & Security*, 135:103509, 2023.

[XZD+23] Binwu Xiang, Jiang Zhang, Yi Deng, Yiran Dai, and Dengguo Feng. Fast blind rotation for bootstrapping fhes. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 3–36. Springer Nature Switzerland, 2023.

[YWH+24] Zewen Ye, Tianyu Wang, Tianshun Huang, Yonggen Li, Chengxuan Wang, Ray C.C. Cheung, and Kejie Huang. HTCNN: High-throughput batch CNN inference with homomorphic encryption. Cryptology ePrint Archive, Paper 2024/1753, 2024.

[Zam22a] Zama. Concrete ML: a privacy-preserving machine learning library using fully homomorphic encryption for data scientists, 2022. https://github.com/zama-ai/concrete-ml.

[Zam22b] Zama. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data, 2022. https://github.com/zama-ai/tfhe-rs.