Neural Cipher Bench: Cost and Portability of Corner-ReLU Cipher Implementations

Siwoo Eum[1], Minwoo Lee[1], Gyeongju Song[1], Seungwon Lee[1], Uijae Kim[1], Doyun Park[1], Minseo Kim[1], Wai-Kong Lee[2], Zhe Liu[3], Anupam Chattopadhyay[4], and Hwajeong Seo[1⋆]

[1] Hansung University, Seoul, Republic of Korea
{shuraatum,minunejip,thdrudwn98,dkajdfhd1,chrisvt424,rhcp030418,msqoo0212,hwajeong84}@gmail.com
[2] Universiti Tunku Abdul Rahman, Kampar, Malaysia
wklee@utar.edu.my
[3] Zhejiang University, Hangzhou, China
zhe.liu@nuaa.edu.cn
[4] Nanyang Technological University, Singapore
anupam@ntu.edu.sg

Abstract. Edge and Internet of Things (IoT) devices run neural networks on GPUs, neural processing units (NPUs), and microcontrollers (MCUs) through machine learning (ML) runtimes. Deep Neural Cryptography (DNC, EUROCRYPT 2026) expressed AES as a bit-exact ReLU network with fixed weights, using a corner-ReLU construction that keeps the cipher inside the affine+ReLU operators such runtimes execute. What this construction costs on edge and IoT hardware, and whether real ML backends execute it correctly, had not been measured. We implement SM4, ARIA, Camellia, and ChaCha20 (spanning substitution–permutation, Feistel, and add–rotate–XOR structures) alongside AES on a shared corner-ReLU core under one construction policy, derive a cost model for width, depth, and neuron count, and measure five execution paths on four platforms. All ciphers admit bit-exact networks whose width and depth diverge with cipher structure, and the model's width predicted where GPU throughput drops. Every configuration passing the correctness check fell below its classical baseline (best 0.80×, SM4 on the GPU), by hundreds to thousands of times for the S-box ciphers on the MCU, and keeping the activations on chip through fusion raised throughput 1.4–3.5× and moved the peak past the predicted cache limit without closing the gap. Some AES configurations on the Apple Neural Engine returned valid-looking but incorrect outputs with no error or warning, a failure reproduced in a minimal graph without project code. Mathematical bit-exactness therefore does not guarantee backend-level bit-exactness, and exact workloads offloaded to edge NPUs require output validation.

⋆ Corresponding author.

Keywords: Benchmarking · Bit-exact execution · Deep neural cryptography · Edge computing · Neural network hardware · Symmetric cryptography

1 Introduction

Standard symmetric-key ciphers are defined over discrete bits and Boolean operations, whereas an ordinary neural network runs as a composition of affine mappings and ReLU over real-valued tensors. Gérault, Hambitzer, Ronen, and Shamir confronted this gap between computational models in *Deep Neural Cryptography* (DNC) [7] at EUROCRYPT 2026. Their two-layer *corner-ReLU construction* assigns ReLU units, called *corner neurons*, to Boolean vertices (input bit patterns) and exactly computes any Boolean function. Composing these networks, they express all of AES-128 as a bit-exact ReLU network with fixed weights, whose output is identical, bit for bit, to a standard implementation rather than a learned approximation. DNC's distinguishing constraint is that the cryptographic functionality does not escape into native integer, bit, or lookup operations but stays inside a standard neural-network graph made of affine mappings and ReLU.

This constraint makes DNC a computational model to be evaluated rather than a performance-optimization device, and it also matches how neural networks are deployed on edge and Internet of Things (IoT) devices, which run models through machine learning (ML) runtimes on integrated GPUs, neural processing units (NPUs), and microcontrollers (MCUs). Such a runtime compiles the model for its hardware and decides where each operation executes, so part of a model can fall back to the host CPU (Section 5.1 shows this for the Apple Neural Engine). Affine layers and ReLU are among the most widely supported operators, so a cipher expressed only in them is a candidate for running inside the same on-device ML pipeline as the model. When cryptographic functionality is to be included as part of a deployable model graph, it must be expressed in the operations the target ML runtime supports, and this paper evaluates the cost and execution correctness of choosing the affine+ReLU construction for that purpose across platforms. Accelerator residency and fewer host calls are possible uses of such a graph, not benefits this paper measures.

Prior work has already shown that ciphers can run efficiently on ML runtimes. TensorCrypt [13] (NDSS 2025) ran AES, ChaCha20, and Salsa20 on the ML runtimes of several platforms, but it executes S-box lookups and bit operations directly with the framework's built-in table-lookup and integer operators, such as `lookup` and `bitAnd`. DNC, by contrast, restricts the admissible operations to affine mappings and ReLU. DNC's restriction is not a functionally stronger abstraction but a more restrictive operator set, so the two approaches answer different questions (Section 2.2). The original paper evaluated only AES, on an NVIDIA A100. This paper asks what is possible, and what cost and failures are observed, when the cryptographic functionality must stay inside a standard affine+ReLU graph.

To answer this question, we implement four standardized ciphers spanning the substitution–permutation network (SPN), Feistel, and add–rotate–XOR (ARX) structures (ARIA, SM4, Camellia, and the table-free ARX contrast case ChaCha20) on a shared corner-ReLU core under one construction policy, and measure all of them on four hardware platforms under a single protocol. The platforms span from a desktop GPU to an IoT-class MCU: an NVIDIA RTX 4080 GPU (CUDA), an x86 AVX-512 laptop CPU, the Apple M4 Pro's integrated GPU (Metal Performance Shaders, MPS) and Neural Engine (ANE), and a Cortex-M4 microcontroller. Over this matrix we ask the following research questions (RQs). *RQ1* asks what the corner-ReLU construction requires for SPN, Feistel, and ARX ciphers under one fixed policy, and whether every instance stays bit-exact. *RQ2* examines how the representation cost (width, ReLU depth, and corner-neuron count) follows from the cipher structure, and how the realized throughput, the gap to the classical baseline, and the observed bottleneck vary with the platform. *RQ3* tests whether real ML backends preserve the bit-exact semantics of the computation graph, and, where they do, how performance still varies with the backend stack (the backend, compiler, and kernel libraries).

The contributions of this paper are as follows.

1. **A common-policy evaluation framework for four ciphers (RQ1).** We implement SM4, ARIA, Camellia, and ChaCha20 as bit-exact ReLU networks on a shared core, where each cipher supplies only its round operations and all four follow one construction policy (one-hot S-boxes, trees of four-input XORs, and ripple-carry addition). We claim no new representability result (Table 1). The comparison under one policy exposes the structure-specific cost axes (one-hot width, XOR accumulation, carry depth).
2. **A predictive cost model and a cross-platform performance map (RQ2).** An analytical cost model computes ReLU depth, width, and corner-neuron count from the round structure. Its width predicts the batch at which GPU throughput drops once the activations outgrow the L2 cache. The measurements dropped at the predicted batch for all four S-box ciphers, and a representation-preserving fusion kernel moved the peak past that batch and raised it by 1.4–3.5×. The best correctness-passing configuration (SM4 on RTX 4080/CUDA) reached 0.80× of its classical baseline, and the S-box ciphers on the Cortex-M4 trailed by factors of hundreds to thousands.
3. **An execution-correctness framework and an NPU counterexample (RQ3).** We frame a corner-ReLU implementation as a chain of conditions whose composition yields a verified output: Boolean exactness, exact numeric encoding over a bounded integer range, and backend semantic preservation. The first two hold by construction (Propositions 1 and 2), and we exhibit a counterexample to the third (Definition 2). On some AES graphs and batches, the Apple Neural Engine silently returned wrong outputs that were still well-formed 0/1 vectors (*backend-level silent data corruption*), a failure that also appeared on a second device and in a minimal graph without project code.

This paper does not propose corner-ReLU ciphers as a practical alternative to classical implementations. AES-NI hardware instructions or the vectorized implementations behind OpenSSL's high-level EVP interface also dominate in our measurements, and the goal is to characterize the cost and correctness of the design space when a representation constraint is present.

Section 2 reviews related work, Section 3 recasts the construction and defines what is measured, Section 4 presents the four constructions and the cost model, Section 5 the protocol and baselines, Section 6 the results, Section 7 the correctness and portability findings, and Section 8 concludes.

2 Related Work

2.1 Bit-Exact Neural Cryptography: Deep Neural Cryptography

The direct foundation of this work is *Deep Neural Cryptography* [7] by Gérault, Hambitzer, Ronen, and Shamir. They introduce the *corner function* technique (Section 3.1) and use it to express AES-128 as a bit-exact ReLU network, including a T-table variant that fuses SubBytes and MixColumns into a single read-out per round. The same paper also treats a protection mechanism (input clipping and output masking) and a key-recovery attack specific to the neural representation, but its empirical benchmark is confined to AES on a single platform, an NVIDIA A100. Part of its motivation is embedding cryptographic functionality inside neural models, which relates to work on undetectable backdoors [10,20]; this paper takes the embedding itself as its deployment setting but does not study hiding or backdoors. Kim's key recovery [14] assumes real-valued oracle access and a corner parameter $c < 1$ (Section 3.1), whereas this paper treats implementation and performance at $c = 1$. This paper continues the technique from the standpoint of implementation and evaluation rather than attack and defense. The generality of the construction is the original paper's contribution, and Table 1 separates the parts this paper reuses from the parts it adds.

2.2 Integer and Indexing Tensor Operators: The Boundary with TensorCrypt

The prior work closest to ours is TensorCrypt [13] (NDSS 2025). Jin et al. present a domain-specific language (DSL) and conversion rules that turn a cipher into a TensorFlow graph, and evaluate AES, ChaCha20, and Salsa20 on GPUs, a cloud TPU, a smartphone, a Raspberry Pi 3, and desktops, with throughput gains over earlier expert-written GPU implementations. TensorCrypt thus already evaluated multiple ciphers on multiple ML runtimes.

The main difference between our work and TensorCrypt is the boundary of the admissible operators. TensorCrypt's DSL admits `bitAnd`, `bitShift`, `slice`, and `lookup` directly as layer types, so S-box and GF(2^8) products are executed by the framework's built-in table-lookup operators and bit operations by its built-in integer operators. This is a tensor-operator program in which ReLU

Table 1. What this paper reuses from DNC [7] and what it adds.

Reused from DNC	
Corner construction	Exact Boolean corners and their composition
AES networks	Natural and T-table designs, here with optimized execution
Protection	Input/output wrapper, here with its overhead measured
Added by this paper	
Four more ciphers	SM4, ARIA, Camellia, and ChaCha20 on a shared core under one policy
Cost model	Width, depth, and corner formulas; cache-drop prediction; policy sensitivity
Platforms	Five execution paths on four platforms, GPU to MCU (DNC: AES on an A100)
Backend correctness	Validation after every transformation; an ANE counterexample

activations never appear, and by the activation-existence criterion of this paper (Section 3.5) it is an execution that leaves the neural representation. The target of DNC and of this paper, by contrast, is a matrix-multiply-plus-ReLU network in which every Boolean operation is computed as a corner-ReLU activation. TensorCrypt showed the benefit of admitting integer and indexing operators, whereas this paper characterizes the observed cost of keeping the corner-ReLU graph across five ciphers and four platforms.

2.3 Learned Neural Cryptography

The learned line trains networks rather than constructing them, beginning with adversarially trained encryption [1] and its successors [4]. Its products are learned approximations with neither standard-cipher compatibility nor cryptographic security guarantees, and ML-based cryptanalysis [9] uses networks to attack ciphers. The network of this work (and of [7]) is instead constructed and matches the standard specification bit for bit.

2.4 Reliability of ML Software and Hardware Stacks

Our correctness finding connects to work on the reliability of ML software and hardware. Deep learning compilers have been shown to contain many bugs across their front ends, optimizations, and back ends [22], including silent correctness bugs that yield wrong outputs without exceptions or warnings [17], and fleet-scale studies report processor cores that silently compute wrong results [11]. On-device ML runs on mobile accelerators whose hardware and software support differ widely across vendors [12]. These studies concern approximate ML

workloads or general computation. A bit-exact cipher turns any such error into a cryptographic failure, which is why Section 5 validates every transformed artifact.

3 Background: Corner Functions and the Definition of a Neural Network Implementation

Here we recast the corner-ReLU construction of the original paper [7] in our notation and formalize the lossless re-encoding property and the definition of what is measured.

3.1 Corner Functions: Bit-Exact ReLU Representation of Boolean Functions

A Boolean function is fully described by its truth table. The corner construction turns that table into a network in the most literal way: every input pattern on which the function outputs 1 gets one ReLU neuron that fires only on that exact pattern, and the function is the sum of those neurons. A neuron dedicated to one input pattern is a *corner neuron*, because each 0/1 pattern is a corner (vertex) of the hypercube $\{0,1\}^n$.

An arbitrary Boolean function $f\colon \{0,1\}^n \to \{0,1\}$ can be represented exactly by a ReLU network with a single hidden layer. For a vertex (corner) v of the hypercube $\{0,1\}^n$, set the weights and bias to

$$w_i = \begin{cases} +1 & v_i = 1 \\ -1 & v_i = 0 \end{cases} \qquad b_v = 1 - \|v\|_1 \tag{1}$$

so that the neuron $\mathrm{ReLU}(w \cdot x + b_v)$ is exactly 1 at $x = v$ and 0 at every other 0/1 input. At $x = v$ the terms with $v_i = 1$ contribute $+1$ each and the others contribute 0, so $w \cdot x = \|v\|_1$ and the pre-activation is $\|v\|_1 + b_v = 1$. At any other 0/1 input at least one bit disagrees with v, and each disagreeing bit lowers $w \cdot x$ by 1, so the pre-activation is at most 0 and the ReLU outputs 0. For example, the two-input AND is 1 only at $v = (1,1)$. Its corner has $w = (+1,+1)$ and $b_v = 1 - 2 = -1$, giving $\mathrm{ReLU}(a + b - 1)$: the output is 1 when $a = b = 1$ and 0 for the other three inputs. This is DNC's corner neuron with the corner parameter $c = 1$, used throughout this paper (a general $c > 0$ divides the weights and the $\|v\|_1$ term of the bias by c). Each corner neuron is thus the indicator of one vertex, and summing the outputs of the corners on which f is true gives

$$f(x) \;=\; \sum_{v\,:\,f(v)=1} \mathrm{ReLU}(w_v \cdot x + b_v) \tag{2}$$

for every 0/1 input. Since a 0/1 input matches exactly one vertex, at most one corner fires on any input and the summation is safe. A multi-output function is obtained by multiplying by a read-out matrix of 0/1 coefficients instead of summing.

With this construction in hand, we fix the terminology used throughout. The *corner-ReLU construction* is the full ReLU representation of Boolean operations, a *corner neuron* is one ReLU unit associated with one Boolean vertex (it outputs 1 for exactly one input bit pattern), and the *corner-neuron count* is the number of such units in the unrolled graph. The vertex indicator a corner neuron realizes is the *corner function* of DNC [7]. When discussing execution, we refer to the runtime values produced by these units as corner activations.

Proposition 1 (Boolean exactness). *With the corner neurons of* (1)*, the read-out* (2) *equals* $f(x)$ *at every binary input, and wiring the* 0/1 *output of one such gadget into the next preserves this, so a cipher composed from them agrees bit for bit with its standard specification.*

Cipher operations map onto this construction as follows. An n-input XOR has 2^{n-1} odd-parity vertices and becomes 2^{n-1} corner neurons (two for key addition, eight for a four-input diffusion XOR), and an XOR with a fan-in (number of inputs) of 5 or more is decomposed into a tree of four-input XORs (XOR4). The AND of Camellia's FL function is a single $\mathrm{ReLU}(a + b - 1)$, and OR is three corner neurons. Rearrangements such as bit permutations, rotations, and ShiftRows are pure indexing that produce no values and consume no parameters or depth.

3.2 One-Hot Expansion of the S-box and T-table Fusion

Expanding a dense 8-bit function such as an S-box into corners per output bit is inefficient: each of the 8 output bits is a Boolean function that is true on roughly half of the 256 inputs, so it would need about 128 corners, or about 1,024 in total. Instead, attaching all 256 corner neurons to the 8 bits of one byte, one per vertex of $\{0,1\}^8$, makes exactly one of them fire, so the ReLU output is the one-hot indicator vector of that byte value (a 256-entry vector with a single 1 at position x). On top of it any 8-bit table T is evaluated by a single linear map. With the read-out matrix $M \in \{0,1\}^{8\times 256}$ whose entry $M_{k,x}$ is output bit k of $T[x]$, $M \cdot \mathrm{onehot}(x) = T[x]$. Not only S-boxes but also constant $\mathrm{GF}(2^8)$ multiplications (the coefficients of AES MixColumns) are tables, so they are linearized the same way. The one-hot layer and its read-out are thus the network's form of a table lookup.

Tables reading the same one-hot can be pre-combined into a composite table $T_2 \circ T_1$ with the same result, so fusing AES's SubBytes and MixColumns into a single read-out is the original paper's T-table implementation. More generally, any weighted combination of read-outs over the same corner activations is a mathematically identical execution rearrangement, and the platform optimizations of Section 6 are all applications of this property.

3.3 The Protection Mechanism

The network is a function on real-valued vectors, not only on bit vectors, and an attacker who can feed fractional inputs observes responses that Boolean inputs

never produce. The original paper [7] turns such queries into a key recovery and counters them by wrapping the network with two ReLU-based layers. An input layer clips each input bit toward 0 or 1 with the ramp $\mathrm{clip}(x) = (\mathrm{ReLU}(x-\epsilon) - \mathrm{ReLU}(x-1+\epsilon))/(1-2\epsilon)$, which is 0 for $x \le \epsilon$, 1 for $x \ge 1-\epsilon$, and linear in between, so bit inputs pass through unchanged. An output mask sums, over all input bits, a bump that is exactly 0 at 0 and 1 and positive in between; the sum is subtracted from every output bit before a final ReLU. A binary query therefore leaves the output untouched, while a non-binary query has every output bit reduced by the mask total, and the output is zeroed entirely once any input bit lies in $[\epsilon, 1-\epsilon]$. The original paper illustrates the threshold with $\epsilon = 1/3$, whereas its public implementation fixes $\epsilon = 1/4$ [8], which is exactly representable in fp16 while $1/3$ is not. We follow the implementation.

The input clipping and the output blanking each add one serial ReLU level, while the mask is computed in parallel, so protection adds exactly 2 to the ReLU depth and changes no interior parameter or corner. These are wrappers at the block input and output, not between rounds (the protected variant covers the four SPN and Feistel ciphers; ChaCha20, the ARX control, is measured unprotected).

3.4 Precision Is Encoding: Lossless Re-encoding

The platforms of Section 5 execute the network in different numeric formats (fp16, fp32, and int8 on the MCU). This subsection shows that the choice cannot change the output. With $c = 1$ the whole network is an integer circuit. Weights are ± 1 or $\{0, 1\}$, biases are integers, and activations are $\{0, 1\}$. The protection wrapper of Section 3.3 is the one exception: its parameters are multiples of $\epsilon = 1/4$, which are likewise exact in fp16 and in a scale-4 integer encoding. Over the operating domain of 0/1 inputs every intermediate stays a small integer, because at most one corner fires per one-hot group and every XOR or gate subcircuit has bounded fan-in (the XOR4 tree of Section 3.1 caps XOR fan-in at four).

Proposition 2 (Lossless re-encoding). *Fix a numeric format and accumulator that represent every integer in the network's operating range exactly and accumulate without rounding. Then evaluating the corner-ReLU network in that format returns the same* 0/1 *output as evaluation over the integers, for every admitted input. In particular, changing among fp16, fp32, and the stated integer encodings does not alter the output.*

Instrumenting all five ciphers under exact arithmetic on binary inputs confirms the premise for the quantities recorded: every layer output, including every corner pre-activation and read-out, stayed within $|v| \le 7$. Partial sums inside an accumulation are bounded separately by the fan-in of the operation: a corner's dot product has at most eight unit terms and a read-out of a one-hot vector has a single nonzero term, so no partial sum exceeds 8 in magnitude. fp16 represents every integer up to $2^{11} = 2048$ exactly, so the choice of fp16, fp32, or int8 is an

encoding question with no accuracy loss by construction. The property concerns accuracy, not speed, and it presupposes that the backend evaluates the graph as written; Section 7 shows one that does not.

3.5 Defining and Bounding What Is Measured

Two things must be fixed before anything is measured: which object is measured, and what it means for a backend to execute that object correctly. Definition 1 fixes the first and Definition 2 the second, and with Propositions 1 and 2 they form a chain: Boolean exactness → lossless encoding → backend preservation → verified output. Without the first definition, successive optimization would silently turn a "corner-ReLU implementation" into a classical one.

Definition 1 (Corner-ReLU cipher implementation). *An implementation is a* corner-ReLU cipher implementation *when every Boolean operation is realized through the corner-ReLU activations introduced by DNC [7], without substitution by native bit operations or value-indexed lookups. The implementation measured in this paper is such a ReLU network obtained by the corner-ReLU construction of Section 3.1–3.2, with its weights and complexity (parameters, ReLU depth, corner-neuron count) frozen, executed by a backend that preserves its semantics in the sense of Definition 2.*

The test we apply to an execution, the *activation-existence criterion*, is a single question: is every corner activation actually computed at run time? If it is, the execution is inside the boundary whatever else it does; kernel fusion (including a ReLU computed inside a matrix-multiply kernel or with integer SIMD), low-precision encoding, graph compilation, keeping activations in registers or shared memory, and compile-time selection among precomputed read-outs are all permitted. If it is not, the execution is outside, because the same result was obtained by another primitive: a table lookup indexed by the state, a native bit operation, or, for a two-input XOR, the single native operation $|a-b|$ in place of its two corners $\mathrm{ReLU}(a-b)+\mathrm{ReLU}(b-a)$. Such a path may well be bit-exact; it is excluded not because it is wrong but because the object measured is then no longer the fixed corner network. The criterion is a scoping rule that keeps every comparison within one family of constructions.

Definition 2 (Backend semantic preservation). *A backend (compiler, runtime, or accelerator driver)* preserves the graph semantics *of a corner-ReLU cipher implementation if, after all of its transformations (compilation, fusion, format conversion, and re-encoding), it returns for every valid input the same output as the exact integer graph of Section 3.1–3.2.*

The first two links of the chain hold by construction (Propositions 1 and 2), but this third one does not, because the backend is outside our control. The property is universal over inputs, so the correctness checks of Section 5.1 (official test vectors and 10^4 random inputs per configuration) can refute it but never establish it. A single failing input suffices to break it, and Section 7.1 exhibits such a counterexample.

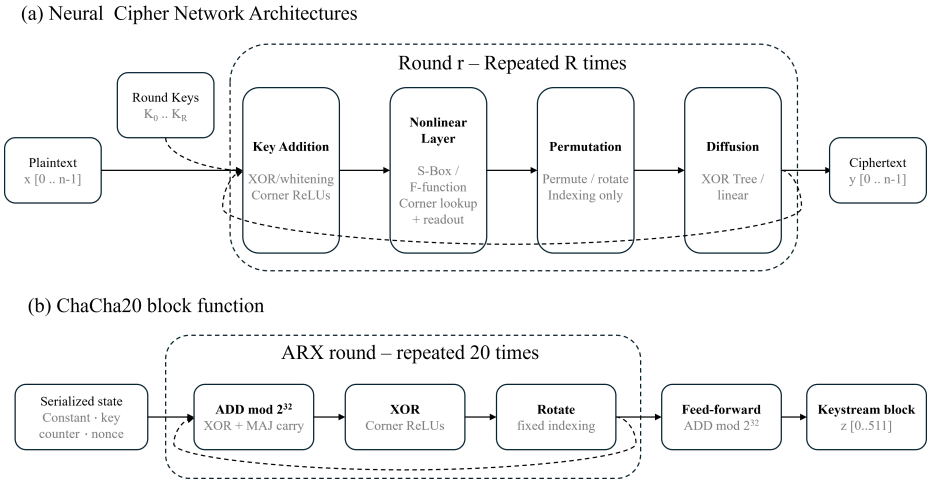

Fig. 1. The two construction templates: (a) the block ciphers (AES, ARIA, Camellia, SM4) and (b) the ChaCha20 block function. Both are bit-exact ReLU constructions with fixed weights; permutation and rotation layers carry no parameters.

4 Corner-ReLU Constructions of the Four Ciphers and a Cost Model

We apply the corner-ReLU construction of Section 3 to four ciphers (SM4, ARIA, Camellia, and ChaCha20) and derive analytical formulas that compute the neural cost from the cipher structure, checked against Table 2 (RQ1). So that differences between ciphers read as cipher structure rather than implementation, all four sit on one shared core (Fig. 1). Each cipher implements only the substitution, permutation, and diffusion hooks, while the corner expansion, XOR neurons, and read-out constructor are common code.

The four ciphers are standardized and were selected to span distinct structural cost drivers rather than for deployment share. AES (FIPS 197 [19]) is the reference SPN with a GF(2^8) S-box. ARIA (RFC 5794 [15]) is a Korean-standard SPN whose diffusion is a pure binary-XOR matrix. SM4 (GB/T 32907 [23,24]) is a Chinese-standard Feistel also registered for TLS cipher suites. Camellia (RFC 3713, CRYPTREC-listed [18,5]) is a Feistel with AND/OR-based FL layers. ChaCha20 (RFC 8439 [21,3]) is a table-free ARX stream cipher used in IETF protocols.

Panel (a) of Fig. 1 is a schematic vocabulary of the block-cipher round, and individual ciphers omit, fuse, reorder, or repeat its stages. Only the stages realized by corners (key addition, one-hot expansion, XOR diffusion) contribute to a round's ReLU depth, since read-out is a linear sum and permutation is indexing. In all four ciphers, rotations, permutations, and Feistel swaps are indexing, every XOR is a corner module, and decryption reuses the network with the decryption round keys. In panel (b), modular addition is a 32-bit ripple-carry circuit whose

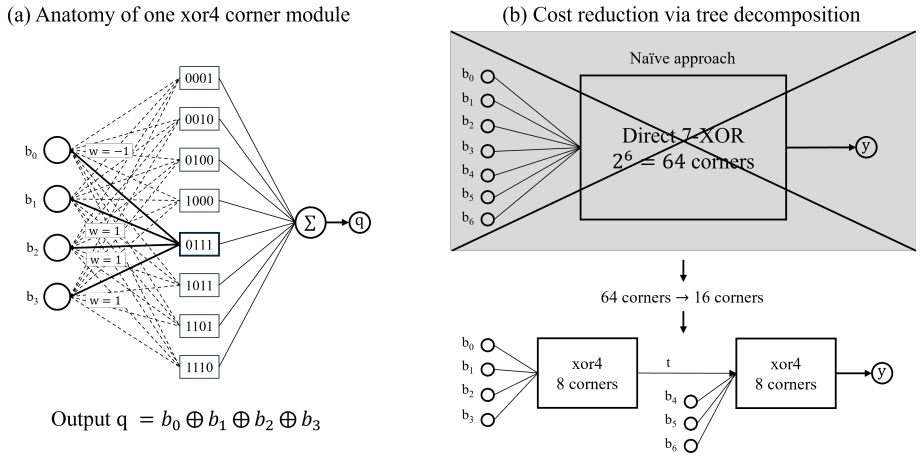

Fig. 2. One XOR4 corner module (a), with the ± 1 weights of the highlighted vertex $v = 0111$, and the fan-in-7 XOR4 tree (b), contrasting direct expansion (64 corners) with tree decomposition (16 corners).

serial carry chain consumes depth 32 per addition. Where an S-box cipher pays its cost in the width of its one-hot layers, this ARX path pays it in depth.

4.1 ARIA: Tree Decomposition of Wide Binary-XOR Diffusion

ARIA is an involutional SPN whose diffusion is a pure binary-XOR matrix rather than a GF(2^8) product [15]. Its four S-box kinds (SB1–SB4) share one 256-corner expansion, since the one-hot depends only on the byte value, and differ only in their read-out matrices.

The design point is the diffusion layer A. The 16×16 binary matrix A has row weight 7 (a single input byte produces seven output bytes, an involution with branch number 8), so each output byte is fixed as the XOR of 7 of the 16 input bytes. Expanding this seven-input XOR into corners directly would need $2^{7-1} = 64$ neurons per bit. Decomposing it into a tree of two four-input XOR (XOR4) modules (the first over four inputs, the second over its output and the remaining three) reduces it to $8 + 8 = 16$ per bit (Fig. 2), a quarter of the corner-neuron count for one more unit of depth, and the same strategy recurs for Camellia (fan-in 5–6) and SM4 (fan-in 5). The per-round depth is four (key addition, substitution, two tree layers), and since the last round omits diffusion but adds one final key addition, the ReLU depth of all 12 rounds is $4R - 1 = 47$.

4.2 SM4: Thin, Deep Rounds

SM4 is a 32-round unbalanced Feistel over four 32-bit words [23]. Each round replaces one word using the other three and a round key,

$$X_{i+4} = X_i \oplus T(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus rk_i),$$

where $T = L \circ \tau$ applies the S-box τ to each of the four bytes and then the diffusion $L(B) = B \oplus (B \lll 2) \oplus (B \lll 10) \oplus (B \lll 18) \oplus (B \lll 24)$. Each part maps onto the construction as follows. The four-input mix $X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus rk_i$ is one XOR4 module per bit, with the round key as its fourth input. The S-box τ is a one-hot expansion of each byte plus a read-out. Rotation is pure indexing, so L costs only the XOR of its five rotated copies, tree-decomposed as an XOR4 module plus a two-input XOR for $8 + 2 = 10$ corners per bit (direct expansion would be 16). The outer $\oplus X_i$ is a two-input XOR corner per bit.

Because T acts on a single 32-bit word, the one-hot expansion of SM4 covers only four bytes per round, a quarter of AES's sixteen, so its S-box stage is narrow. Each round, however, has five serial corner stages (the input-mix XOR4, the one-hot expansion, the two tree layers of L, and the outer XOR), and 32 rounds give a depth of $5 \times 32 = 160$, over five times that of the AES T-table path. SM4 is thus thin but deep, the opposite profile from AES.

4.3 Camellia: Boolean-Gate Corners and the FL Layer

Camellia-128 is an 18-round Feistel over two 64-bit halves with key whitening at the input and output, and after rounds 6 and 12 a key-dependent layer applies FL to one half and its inverse FLINV to the other [18]. Each round computes

$$L_r = R_{r-1} \oplus F(L_{r-1}, k_r), \qquad R_r = L_{r-1},$$

where $F(X, k) = P(S(X \oplus k))$ XORs the round key into the half, passes each of its eight bytes through one of four S-boxes s_1–s_4, and applies the byte-XOR diffusion P, in which every output byte is the XOR of five or six input bytes. The mapping is the one already used for ARIA. The key XOR is a two-input XOR corner per bit, the eight S-box lookups share a single one-hot expansion and differ only in their read-outs, P is tree-decomposed to 12 or 10 corners per bit, and the Feistel XOR is a two-input corner. A round thus has five serial corner stages (key XOR, one-hot expansion, the two tree layers of P, and the Feistel XOR).

FL is the one Camellia component that fits neither the XOR nor the S-box mold. On a half $X = (X_L, X_R)$ with subkey (k_L, k_R),

$$Y_R = ((X_L \wedge k_L) \lll 1) \oplus X_R, \qquad Y_L = (Y_R \vee k_R) \oplus X_L,$$

and FLINV is its inverse. The AND and OR are handled by the corner expansion of an arbitrary Boolean function (Section 3.1): AND has only one true vertex and is a single $\mathrm{ReLU}(a + b - 1)$ neuron, and OR has three true vertices and is three corner neurons, with the key bit as the second operand of each gate. The four serial gates of FL (AND, XOR, OR, XOR) give it depth 4, so the total is $1 + 5 \times 18 + 4 \times 2 + 1 = 100$ (input whitening, 18 F-rounds of depth 5, the two FL/FLINV layers, and output whitening).

4.4 ChaCha20: The Depth Cost of the Ripple-Carry Construction

The ARX contrast case ChaCha20 [21,3] uses neither an S-box nor XOR-matrix diffusion, only 32-bit modular addition, XOR, and rotation. The network computes the block function that turns the 512-bit initial state (constants, key, counter, nonce) into a keystream block. Encryption and decryption are the same keystream generation, and the byte XOR with the plaintext is handled on the host. XOR is a two-input corner and rotation is indexing (Section 3.1), so the only new element is the mod 2^{32} addition. We build it as a ripple-carry circuit, which computes the carry bit by bit from the least significant position, each carry from the one before. Ripple-carry is the chosen construction, not the only one. It maps onto small local corner gadgets and keeps the activation width small, whereas a carry-lookahead adder would cut the serial depth but raise width and corner-neuron count. At bit position p the sum bit is $s_p = a_p \oplus b_p \oplus c_p$ (a three-input XOR, four corners) and the carry is $c_{p+1} = \mathrm{MAJ}(a_p, b_p, c_p)$ (the majority function, four corners), evaluated in parallel at the same depth. Because the carry chain is serial, a single 32-bit addition consumes a ReLU depth of 32.

The quarter-round (QR) amplifies this cost. On four words it performs

$$a \mathrel{+}= b;\ d \mathrel{\oplus}= a;\ d \lll 16; \quad c \mathrel{+}= d;\ b \mathrel{\oplus}= c;\ b \lll 12;$$
$$a \mathrel{+}= b;\ d \mathrel{\oplus}= a;\ d \lll 8; \quad c \mathrel{+}= d;\ b \mathrel{\oplus}= c;\ b \lll 7,$$

and each step depends on the one before, so one QR chains four additions and four XORs for depth $4 \times 32 + 4 = 132$ (the rotations cost nothing). A round applies four QRs in parallel, the 20 rounds are serial, and the final feed-forward adds the initial state to the result, giving $20 \times 132 + 32 = 2{,}672$. ChaCha20 thus has the smallest stored-parameter count (41: the key enters as part of the input state, so only the gate weights are stored) yet the largest corner-neuron count (over 100,000; a single 32-bit addition alone spends 247, eight per bit position less a half-adder at bit 0 and no carry-out at bit 31) and a ReLU depth 89 times the AES T-table path. The representation passes the bit-exact check, and Section 6 reports the cost of a serial depth in the thousands.

4.5 Cost Formulas from Cipher Structure

The values of Table 2 are also derivable analytically from the specification. The formulas below share the construction rules of our implementation, so their agreement with the table is a consistency check rather than independent verification. Under this paper's fixed policy (XOR4 trees for fan-in ≥ 5, ripple-carry addition, cross-round weight sharing, and T-table fusion where possible), they predict the cost from the specification before implementing, and a different choice yields different formulas.

(i) **S-box corner neurons and width.** With B bytes expanded per round over R rounds, the unrolled S-box corner-neuron count is $C_{\mathrm{sbox}} = R \times B \times 256$, matching the corners column minus its XOR contribution. The *concurrent*

Table 2. Hardware-independent complexity per block, fully unrolled. Natural: unfused construction; TTables: T-table fusion (AES encryption only; decryption uses the Natural inverse path with identical metrics). params: stored weights, shared weights once. $\text{depth}_{\text{ReLU}}$: serial ReLU layers, plain and with the protection wrapper (prot.). MACs: over all connections (dense) or nonzero activations only (nz). corners: corner-neuron count.

cipher	impl.	params	$\text{depth}_{\text{ReLU}}$		$\text{MACs}_{\text{dense}}$	MACs_{nz}	corners
			plain	prot.			
AES-128	Natural	9,902	39	41	1,582,592	1,123,840	89,856
	Natural (Dec)	13,998	39	41	2,172,416	1,418,752	89,856
	TTables	9,902	30	32	1,287,680	828,928	52,992
	TTables (Dec)	13,998	39	41	2,172,416	1,418,752	89,856
SM4	Natural	5,422	160	162	598,016	466,944	53,248
ARIA-128	Natural	12,206	47	49	883,200	686,592	75,008
Camellia-128	Natural	12,234	100	102	648,192	500,736	55,680
ChaCha20	Natural	41	2,672	—	288,928	288,928	103,472

activation width of one substitution stage is $W_{\text{sbox}} = B \times 256$, without the round factor, because the rounds are serial and only one stage's corners are live at a time.

(ii) **XOR diffusion.** An n-input XOR is 2^{n-1} corners per bit, and fan-in ≥ 5 is decomposed with an XOR4 tree to 16, 12, or 10 per bit (fan-in 7, 6, 5), at one depth unit per stage.

(iii) **Depth.** (round count) $\times$ (serial corner stages per round) plus a boundary correction. Wide shallow SPNs have few stages (ARIA 4), thin Feistels many (SM4, Camellia 5), and the round count dominates (ARIA 47, SM4 160).

(iv) **ARX depth.** Ripple-carry mod 2^w addition costs serial depth w, so ARX depth grows as (serial additions per round) $\times w$, and ChaCha20's is 2,672.

(v) **Stored parameters.** The count is (S-box read-out tables) $\times(8 \times 256)$ + one-hot (256×9) + gate constants, plus round-key storage, the only term that scales with the round count.

The width W_{sbox} of formula (i) instantiates as $16 \times 256 = 4{,}096$ for AES and ARIA (whole state in parallel), 2,048 for Camellia's F-half, 1,024 for SM4, and a few hundred gate corners for ChaCha20. With 2-byte fp16 activations, a batch of b blocks holds $2bW$ bytes of one-hot activations, so the model predicts that they reach the RTX 4080's 64 MB L2 cache at batch 8,192 for AES and ARIA, 16,384 for Camellia, and 32,768 for SM4. Section 6.2 tests this prediction against the measurements.

The formulas also show how much the reported costs depend on the policy. Expanding each wide XOR directly, without the tree of formula (ii), would lower the depth of SM4, Camellia, and ARIA by about 18–23% but raise their corner counts by about 11%, 27%, and 90%, and ARIA's diffusion stage would then be wider (8,192) than its S-box stage, moving its predicted cache limit to batch 4,096

(AES and ChaCha20 have no XOR of fan-in 5 or more). We did not measure this alternative. Because it raises the corner count that dominates the MCU cost and, for ARIA, the width that sets the GPU cache limit, we do not expect it to change the shortfall conclusions of Section 6, although the magnitudes would shift.

Applying the construction to the four ciphers required no new universality theorem, only fan-in management and direct Boolean-gate expansion. Under one policy the main cost driver is the cipher structure, a statement about the chosen one-hot, XOR4 tree, and ripple-carry family that implies no optimality or lower bound over all ReLU circuits.

5 Benchmark Protocol and Baselines

5.1 Protocol and Hardware

Every configuration is measured with the same protocol on all platforms. The batch size sweeps 12 steps from 1 to 131,072 (1, 64, 128, 512, and then powers of two). Each configuration is preceded by 50 warm-up iterations and executed 100 times, asynchronous accelerator execution is synchronized to completion before the time is taken, and throughput in Mbps is computed from the mean execution time, with peak throughput the maximum over the sweep. Before timing, every implementation passed the official known-answer tests of its standard [19,2,15,23,18,21], and after every backend transformation (compilation, Core ML conversion, MCU int8 encoding) its output was compared bit for bit with the reference implementation on 10^4 random inputs; configurations that failed were excluded from the performance results. In the main performance map (Table 4) the relative standard deviation of the peak is at most 0.4% except on the four-threaded AVX-512 CPU path, where scheduling noise raises it to 35%, so CPU-path ratios are read only at order-of-magnitude scale.

The RTX 4080 (16 GB, 64 MB L2 cache, driver 535.309.01) is hosted by an Intel Core i7-13700KF with 32 GB of memory under Ubuntu 24.04. The AVX-512 CPU is a four-core Intel Core i7-1165G7 (Tiger Lake, 2.8 GHz base clock) laptop. The Apple platform is an M4 Pro with a 16-core Neural Engine under macOS 26.5.1, whose models are converted with coremltools 9.0 and executed by the Core ML runtime of that macOS release. The MCU is an STM32L4+ Cortex-M4 at 48 MHz with 2 MB of flash and 640 KB of SRAM. Table 3 lists the software stack of each path.

Two platforms constrain the sweep. On the MCU a memory limit makes batch 1 the only point, which does not affect the reading because batching there only amortizes call overhead. On the Neural Engine the execution location differs per batch, and a Core ML `CPU_AND_NE` designation alone does not guarantee ANE execution, so we admit only batch configurations whose Neural Engine residency (the share of the estimated compute cost that the Core ML compute plan assigns to the Neural Engine) is at least 0.95. Batch configurations whose model failed the correctness check (Section 7.1) were not timed directly. Where possible, we

Table 3. Measurement conditions per (platform, backend). The tables in Section 6 reference these rows as Cn.

#	platform/backend	device	stack	thr.
C1	apple_m4/ane	Apple M4 Pro (Neural Engine)	coremltools 9.0 + torch 2.12.1	1
C2	apple_m4/mps	Apple M4 Pro (integrated GPU)	torch 2.12.1	1
C3	apple_m4/openssl-evp	Apple M4 Pro (CPU)	OpenSSL 3.6.3	1
C4	cortex_m4/int8-dense	STM32L4+ Cortex-M4	gcc 15.2.1	1
C5	cortex_m4/int8-dense-dsp	STM32L4+ Cortex-M4	gcc 15.2.1	1
C6	cortex_m4/mbedtls	STM32L4+ Cortex-M4	GmSSL / mbedTLS v3.6.6	1
C7	cpu_simd/onednn	Intel i7-1165G7 [AVX-512]	torch 2.12.1+cpu	4
C8	cpu_simd/openssl-evp	Intel i7-1165G7	OpenSSL 3.6.3	1
C9	cpu_simd/refpy	Intel i7-1165G7	torch 2.12.1+cpu	4
C10	cpu_simd/refpy-fp32	Intel i7-1165G7	torch 2.12.1+cpu	4
C11	cpu_simd/torch-cpu-avx512-fp32	Intel i7-1165G7 [AVX-512]	torch 2.12.1+cpu	4
C12	cpu_simd/torch-cpu-fp16	Intel i7-1165G7 [AVX-512]	torch 2.12.1+cpu	4
C13	nvidia/cuda	RTX 4080	torch 2.12.1+cu126	1
C14	nvidia/openssl-evp	Intel i7-13700KF	OpenSSL 3.6.3	1

recovered the same workload by repeating the largest passing smaller batch (for example, batch 4,096 as four dispatches of the batch-1,024 model), marked these records as split-batch results, and do not use them as evidence that the original model is correct.

5.2 Baselines and the Semantics of Comparison

The baseline for each cipher is the classical software implementation in practical use on that platform. OpenSSL is fixed at the single version 3.6.3, built natively on each platform, and its EVP interface is measured with the same protocol. EVP automatically uses the hardware path the platform supports, so the AES baseline is the AES-NI path on x86 and the ARMv8 Cryptography Extensions (ARMv8-CE) path on Apple. The baseline for the Cortex-M4 uses only citable reference implementations (mbedTLS, and GmSSL for SM4) rather than a hand-written C implementation. The comparison mode is electronic codebook (ECB), whose independent blocks match the parallelism of the batched neural network, and throughput is normalized to Mbps. For ChaCha20 the neural side generates the keystream, while the EVP baseline also XORs it with the plaintext, so the baseline does strictly more work and the shortfall conclusion is conservative.

Baseline strength is asymmetric across ciphers. AES faces hardware instructions and ChaCha20 vectorized SIMD, whereas ARIA and Camellia face table-based software. This asymmetry is part of the result and shapes the ratio column of Section 6.

Throughout this paper the neural/baseline ratio is called not a "speedup" but the *ratio of independently measured peak throughputs*: the neural peak over its batch sweep divided by the baseline peak over its own sweep. The two peaks generally occur at different batches, and Table 4 lists both. As Table 3 shows, the two measurements differ in execution location (accelerator vs. host CPU), thread count (single-threaded EVP vs. four-threaded CPU neural network), and transfer scope (only the ANE includes input and output copying). The ratio therefore compares deployment configurations, not kernels on the same hardware; it is neither a comparison with GPU-native cipher kernels [16,6] nor, since

no experiment swaps the corner-ReLU graph for a native graph on the same device, a measurement of the ReLU-only restriction alone.

6 Results: The Performance Map and Observed Bottlenecks

The RTX 4080 CUDA path is the *reference platform* that carries the full batch sweep and the mechanism analyses, while the other three platforms test whether the trends found there carry over.

6.1 The Performance Map: The Gap per Deployment Configuration

Table 4, the result over the four main-performance paths, shows a consistent shortfall. No entry passing the correctness check exceeds the classical baseline. (The M4 Pro GPU entries peak at batch 131,072, the end of the sweep, so they are sweep maxima rather than demonstrated global peaks.) The best entry is SM4 on the RTX 4080 CUDA path (0.80×), and the gap widens steeply by platform and cipher, reaching hundreds to thousands of times for the S-box ciphers on the MCU.

The gap is also structured, not random. The same cipher's ratio varies by up to roughly 730× across platforms, and within a platform it depends on the kind of baseline. AES shows the lowest ratio wherever its baseline is a hardware instruction (ARMv8-CE, AES-NI), not because neural AES is uniquely slow, and SM4, ARIA, and Camellia, with no dedicated instruction, generally rank better. SM4 is the best entry on CUDA, MPS, and AVX-512, consistent with its one-hot width (1,024), a quarter of that of AES and ARIA, which keeps its activations in cache up to larger batches (Section 6.2). On the RTX 4080 the lowest ratio is instead ChaCha20, whose depth 2,672 meets a highly vectorized SIMD baseline. On the scalar Cortex-M4, where the cost lies in the corner computations (Section 6.4), ChaCha20 has the highest ratio, consistent with its having the fewest corners per output byte (its 64-byte block spreads the corners over four times as many bytes).

Table 5 reports the Apple ANE path separately. Its timing includes input and output copying, because Core ML's predict call takes its inputs from and returns its outputs to host memory, with no way to keep them resident on the accelerator between calls, and its reference sweep stops at batch 4,096. An exploratory AES run up to batch 32,768 stayed flat at 4.3–4.6 Mbps, close to the reference peaks, while larger batches fell below the residency threshold or failed to convert. The ANE's valid configurations were also below the baseline, and separately, the path was not always correct (Section 7.1).

Fig. 3 is the batch dimension of this map. The reference platform (a) descends after its peak, the CPU (c) shows the same shape at a much earlier batch, and only the M4 Pro's GPU/MPS path (b) saturates within the sweep range, while the gap to the baseline holds across the whole sweep.

Table 4. Performance map: peak throughput and ratio to the classical baseline per (cipher × platform), encryption, plain, 100 repetitions. Cn denotes a row of Table 3; the measurement conditions behind the ratio are given in Section 5.2.

		neural (best)			baseline (best)			
cipher	platform	Mbps	batch	impl./prec.	Mbps	batch	impl.	ratio
AES-128	M4 Pro GPU^{C2}	**96.2**	131,072	Natural, fp16	199,266	32,768	aes-hw^{C3}	0.00048×
	Cortex-M4^{C5}	**0.00146**	1	TTables, int8	3.3	1	swref^{C6}	0.00045×
	AVX-512 CPU^{C7}	**7.7**	512	TTables, fp32	110,725	65,536	aes-ni^{C8}	0.000070×
	RTX 4080^{C13}	**1,643**	16,384	TTables, fp16	133,388	131,072	aes-ni^{C14}	0.012×
SM4	M4 Pro GPU^{C2}	**228**	131,072	Natural, fp16	1,765	2,048	evp^{C3}	0.129×
	Cortex-M4^{C5}	**0.00406**	1	Natural, int8	3.7	1	swref^{C6}	0.0011×
	AVX-512 CPU^{C7}	**7.5**	512	Natural, fp32	1,171	32,768	evp^{C8}	0.0064×
	RTX 4080^{C13}	**1,341**	32,768	Natural, fp16	1,673	16,384	evp^{C14}	0.802×
ARIA-128	M4 Pro GPU^{C2}	**131**	131,072	Natural, fp16	2,641	2,048	evp^{C3}	0.050×
	Cortex-M4^{C5}	**0.00285**	1	Natural, int8	3.0	1	swref^{C6}	0.00094×
	AVX-512 CPU^{C7}	**2.3**	128	Natural, fp32	1,435	8,192	evp^{C8}	0.0016×
	RTX 4080^{C13}	**645**	8,192	Natural, fp16	2,057	32,768	evp^{C14}	0.314×
Camellia-128	M4 Pro GPU^{C2}	**170**	131,072	Natural, fp16	2,664	32,768	evp^{C3}	0.064×
	Cortex-M4^{C5}	**0.00363**	1	Natural, int8	2.8	1	swref^{C6}	0.0013×
	AVX-512 CPU^{C7}	**3.1**	512	Natural, fp32	2,100	4,096	evp^{C8}	0.0015×
	RTX 4080^{C13}	**679**	32,768	Natural, fp16	2,657	32,768	evp^{C14}	0.256×
ChaCha20	M4 Pro GPU^{C2}	**171**	131,072	Natural, fp16	27,230	8,192	evp^{C3}	0.0063×
	Cortex-M4^{C5}	**0.0429**	1	Natural, int8	2.3	1	swref^{C6}	0.019×
	AVX-512 CPU^{C7}	**4.2**	8,192	Natural, fp32	56,658	32,768	evp^{C8}	0.000074×
	RTX 4080^{C13}	**83.1**	4,096	Natural, fp16	35,394	131,072	evp^{C14}	0.0023×

6.2 The Cause of the Peak: The Capacity-Bound Hypothesis and a Vertical-Fusion Intervention

Every optimization in this section and the next changes only the execution and keeps the activations computed (Section 3.5). The location of the throughput peak is explained by cache capacity, not asymptotic saturation. The corner one-hot activations grow as (batch) × (concurrent activation width, Section 4.5) × (element size), and once this tensor exceeds the RTX 4080's 64 MB L2 cache the read-out matrix multiply becomes limited by DRAM bandwidth and throughput drops sharply, a drop we call the *cliff*. Within the resolution of the doubling sweep, the measurements before fusion dropped at the batches predicted from the width in Section 4.5 (8,192 for AES and ARIA, 16,384 for Camellia, and 32,768 for SM4; Table 6).

We tested this with a vertical-fusion intervention. In the profile, `torch.compile` writes a (batch, 256) one-hot tensor to device memory between the corner and read-out matrix multiplies, and the separate ReLU kernel takes 19% (batch 4,096) to 41% (batch 16,384) of the AES T-table forward pass. The vertical-fusion kernel (Triton) fuses corner-affine, ReLU, and read-out into one kernel and keeps the one-hot only in registers and shared memory, preserving activation count, width, and depth.

Table 6 shows the effect. The gain of fusion grows with batch and is largest beyond each cipher's predicted L2 limit, the peak batch moved outward (AES

Table 5. Peak throughput of the Apple M4 Pro Neural Engine (configurations with Neural Engine residency ≥ 0.95 that pass the correctness check).

	neural (best)[C1]			baseline (best)			
cipher	Mbps	batch	impl./prec.	Mbps	batch	impl.	ratio
AES-128	**5.0**	512	TTables, fp16	199,266	32,768	aes-hw[C3]	0.000025×
SM4	**9.4**	1,024	Natural, fp16	1,765	2,048	evp[C3]	0.0053×
ARIA-128	**7.7**	128	Natural, fp16	2,641	2,048	evp[C3]	0.0029×
Camellia-128	**4.9**	128	Natural, fp16	2,664	32,768	evp[C3]	0.0018×
ChaCha20	**7.7**	512	Natural, fp16	27,230	8,192	evp[C3]	0.00028×

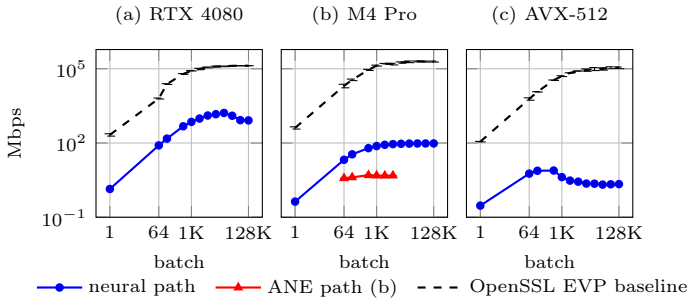

Fig. 3. Representative batch sweep curves for (a) CUDA, (b) MPS/ANE, and (c) CPU (AES-128, encryption, plain; dashed: each machine's OpenSSL EVP baseline).

T-table from 4,096 to 16,384), and the peak rose by 1.44–3.45× depending on the cipher. Before fusion the AES T-table throughput fell from 559 Mbps at batch 4,096 to 184 Mbps at 65,536; after fusion the same-session value at 65,536 is 4.53× higher, and the post-fusion curve (Fig. 3a) shows no drop at the predicted limit. Fusion also changes launch count and scheduling, so this does not prove a single cause, but the width-based prediction, the profiled round-trip cost, and the batch-dependent gain together support the one-hot memory round trip as the main bottleneck.

On the M4 Pro's unified memory the transition from the system-level cache (SLC) to DRAM is far gentler than the RTX 4080 cliff (a per-kernel bandwidth drop of about 1.7× versus 5× or more). The AVX-512 CPU, with its small caches, and the ANE path both peak early, near batch 512. These shapes are consistent with the same memory-tier model, though the paths also differ in stack (Section 7.2), so this is a consistency check, not a controlled verification.

6.3 A Closer Look at the Reference Platform (RTX 4080/CUDA)

Encoding Every int8 and fp8 (E4M3 format) configuration passed the correctness check bit for bit, as Section 3.4 predicts, but fp16 gives the best peak for every

Table 6. Vertical fusion on the RTX 4080 (encryption, fp16, same-session medians): gain over pre-fusion execution by batch, the L2 limit predicted from the width, and the peak batch before and after.

		gain at batch					peak batch	
cipher	L2 limit	4,096	8,192	16,384	32,768	65,536	before	after
AES Natural	8,192	2.87×	7.06×	10.48×	7.96×	6.81×	4,096	16,384
AES T-table	8,192	2.26×	5.45×	7.05×	5.57×	4.53×	4,096	16,384
SM4	32,768	1.62×	1.87×	1.80×	4.15×	5.09×	16,384	65,536
ARIA	8,192	1.44×	3.46×	3.91×	2.97×	2.68×	4,096	8,192
Camellia	16,384	1.36×	1.34×	2.94×	3.37×	2.97×	8,192	32,768

cipher. The int8 path, whose nominal Tensor Core throughput is twice as high, is slower because the tested PyTorch integer matrix-multiply path returns 32-bit accumulators that must be converted back before the next layer, which cancels the nominal gain.

Read-out fusion Read-out fusion concatenates the read-out matrices reading the same one-hot into one (Section 3.2), which keeps the corner graph and its MAC count unchanged yet raised the peak by +48–90%, so the bottleneck is memory traffic, not computation.

Comparison against the original implementation On the same hardware and stack, this paper's optimized AES implementation is 11.4–12.7× faster than the Python implementation the original paper released (refpy [8]). The construction is unchanged and only the execution differs, so not all of the shortfall in Table 4 is fixed-graph cost.

6.4 When There Is No Matrix Engine: An MCU Cost Breakdown

At the opposite extreme, on the Cortex-M4's 48 MHz scalar core with no matrix engine or wide SIMD (the 32-bit SMLAD dual multiply-accumulate instruction is the only parallel primitive), the neural implementation trails the reference implementations (mbedTLS, GmSSL) by hundreds to thousands of times on the S-box ciphers and tens of times on ChaCha20 (Table 4), measured with the int8 dense kernel with DSP instructions (C5) at batch 1.

By formula (i) of Section 4.5, the corners column of Table 2 splits exactly into one-hot-expansion corners plus residual diffusion and gate corners (40,960 + 12,032 AES T-table, 32,768 + 20,480 SM4, 49,152 + 25,856 ARIA, 36,864 + 18,816 Camellia, 0 + 103,472 table-free ChaCha20), and the MCU path computes both classes in full. Substitution that classical code performs as one table load per byte runs as 256 corner evaluations plus a read-out, and diffusion that classical code finishes in a few native XOR instructions runs as tens of thousands of

Table 7. Apple Neural Engine silent corruption, AES encryption, fp16, `CPU_AND_NE` unless noted. Wrong blocks are given out of the blocks checked; "fail" marks cells whose count was not recorded.

Device	torch, exporter	Constr.	Batch	Wrong blocks
M4 Pro	2.12.1, torch.export	Natural	1,024	fail
M4 Pro	2.12.1, torch.export	T-table	2,048; 4,096	fail
M4	2.2.2, jit.trace	T-table	2,048	16 / 2,048
M4	2.2.2, jit.trace	T-table	4,096	24 / 4,096
M4	2.12.1, jit.trace	T-table	2,048	fail
M4	2.12.1, torch.export	Natural	1,024	64–80 / 10,000
M4	2.12.1, torch.export	T-table	2,048; 4,096	64–80 / 10,000
M4	same models, `CPU_ONLY`	both	all	0
M4	2.12.1, fused read-out	T-table	2,048; 4,096	0

corner accumulations, so the shortfall is the corner arithmetic itself. Consistent with this decomposition, vectorizing only the one-hot expansion with the DSP instructions (all 256 corners still evaluated) raises throughput by 1.24–1.51×, and ARIA, whose baseline is weak and whose diffusion is pure XOR, was the most plausible candidate to overtake its baseline, but it did not, and neither did any other cipher. ChaCha20, which has no one-hot at all, computes every corner one stage at a time along its depth-2,672 chain, where a matrix engine would absorb them into matrix multiplies. In the evaluated construction and on this Cortex-M4 implementation, the corner-ReLU form therefore does not provide practical throughput, and the decomposition attributes the cost to the corner computations the graph requires rather than to a single missing optimization. Other constructions and kernels were not measured.

7 Discussion (RQ3): Portability

A backend does not necessarily preserve bit-exact semantics (Section 7.1), and where it did, hardware features alone did not predict performance (Section 7.2).

7.1 RQ3-A: NPU Silent Data Corruption

On the Apple Neural Engine path (Core ML `CPU_AND_NE`), specific AES-encryption configurations returned wrong 0/1 outputs without any error or warning, while decryption and the other four ciphers remained correct (Table 7). In the reference run on the M4 Pro (converted with coremltools 9.0 and executed by Core ML on macOS 26.5.1), the failing batches were 1,024 for the Natural construction and 2,048 and 4,096 for the T-table construction. The dependence on batch is non-monotonic (Natural batches 2,048 and 4,096 pass), which precludes a simple prediction rule, and models with the key baked into the weights and models taking the round keys as inputs failed at the same cells.

The same failure family had first appeared on an earlier M4 system with the same coremltools and macOS versions, where we examined it further.

The same exported models matched the reference implementation bit for bit under `CPU_ONLY`, so one model gives different results depending on the execution path. On the earlier M4 the failure survived two PyTorch versions (2.2.2 and 2.12.1), two exporters (`jit.trace` and `torch.export`), reruns, and fresh exports. The failures were sparse (Table 7). The same blocks came out wrong on every rerun while the wrong bits varied, and the outputs stayed well-formed 0/1 vectors. The compute plans of failing and passing batches were identical. Neither the export path nor operator placement alone therefore explains the failure.

To isolate the trigger, we built a minimal reproducer of 269 lines from the structure of the AES graphs, with no project code. It fails only when six elements co-occur: a 4D reshape and transpose, one shared Linear+ReLU expansion, three read-outs from it with two of them rotated by different offsets, stacking followed by a reduction, and a batch above a threshold. Fusing the read-outs, removing the rotation, summing instead of stacking, or a flat layout each makes it pass. These observations narrow the trigger to a graph pattern on the `CPU_AND_NE` path, but they do not attribute the cause to the converter, compiler, scheduler, or hardware, and they are specific to the tested devices and software versions. We reported the issue to Apple (Feedback FB23667767).

Silent corruption breaks cryptographic correctness, and mathematical equivalence does not establish that an approximation-oriented backend preserves bit-exact execution. The corrupted outputs are still valid 0/1 vectors, so without a check they cannot be told apart from correct outputs. A graph-level workaround, fusing the three T-table read-outs into one, removed the failure at batches 2,048 and 4,096, yet it lowered the Neural Engine residency to 0.925, below the 0.95 threshold, so part of the work would run on the CPU. For IoT deployments that move exact integer workloads such as cryptography onto on-device NPUs, output validation therefore belongs in the deployment pipeline, not only in benchmarking.

7.2 RQ3-B: Hardware Features Alone Cannot Predict Performance

Even where semantics were preserved, performance portability was not automatic.

First, precision support does not imply a fast path. The Tiger Lake CPU has AVX-512 but not AVX-512-FP16 and torch lacks a vectorized fp16 kernel, so fp32 is 3.1–12.5× faster than fp16 at no accuracy cost (Section 3.4), a combined hardware–stack limitation.

Second, the same optimization pays differently across stacks. Read-out fusion raised the CUDA peak by +48–90% but the MPS peak on the earlier M4 system by only +2%, because the MPS compiler had already fused it. CUDA was limited by the one-hot round trip, while MPS was limited by diffusion split into small kernel launches (over 85% of forward time) and improved only after whole-round fusion, which keeps the graph semantics and reduces only the number of kernels.

Third, the MPS ordering of the ciphers followed the execution mode. ChaCha20 (depth 2,672) beat AES (depth 39) in eager (per-op) mode on the earlier M4 because of launch overhead, AES led under `torch.compile` on the M4 Pro, and ChaCha20 leads again with the whole-round fusion of the reference record (Table 4).

7.3 The Cost of Protection and Key Agility (Additional Observations)

The block-boundary protection slowdown was within 5% on CUDA (0.99–1.05×), ANE (0.99–1.04×), and Cortex-M4 (1.00×), while MPS reached 1.04–1.19× with its protected peak at a smaller batch (AVX-512 was excluded for its differing fusion conditions). Rekeying took under 1 ms to a few ms on the torch paths, and NPU block ciphers with a baked-in key needed a re-export, reduced to sub-millisecond by taking the round keys as inference inputs; ChaCha20, whose initial state is a network input, needs none. Devices that change session keys often should therefore take keys as model inputs rather than compile them into the model.

Although the source graph has data-independent control flow and no value-indexed lookup, we evaluate no physical side-channel or fault attacks, and the ANE corruption of Section 7.1 is an incorrectness observation, not a demonstrated fault-attack primitive.

8 Conclusion

We implemented SM4, ARIA, Camellia, and ChaCha20 alongside AES as bit-exact corner-ReLU networks under one construction policy and measured them from a desktop GPU to a Cortex-M4 MCU. All five graphs passed validation (RQ1). Width, depth, and corner-neuron count follow from cipher structure under the chosen policy, and the width predicted where GPU throughput drops. Every correctness-passing configuration trailed its baseline (best SM4 on RTX 4080/CUDA, 0.80×), and fusion of the one-hot memory round trip raised the peak 1.4–3.5× without changing graph width or depth (RQ2). One Apple Neural Engine stack produced silent data corruption, and optimization gains varied across correct stacks (RQ3).

This paper does not propose corner-ReLU ciphers as a practical alternative, and native cryptographic instructions or libraries remain the better choice for edge and IoT designers wherever they are available. Where a deployment requires the cipher to stay inside an affine+ReLU graph, the width and depth formulas estimate its memory footprint and serial depth before implementation, and the MCU results show that without a matrix engine the cost is prohibitive.

Every backend transformation, particularly onto an NPU, requires output validation, because mathematical bit-exactness does not guarantee backend-level bit-exactness. The results map feasibility and cost within the evaluated construction family, with no claim of advantage over native tensor graphs, optimality over

ReLU circuits, or security. Remaining work includes native baselines on the same runtime, alternative circuits and activations, energy measurements on MCUs and NPUs, and other NPUs and stack versions.

References

1. Abadi, M., Andersen, D.G.: Learning to protect communications with adversarial neural cryptography (2016), arXiv:1610.06918
2. Bassham III, L.E.: The advanced encryption standard algorithm validation suite (AESAVS). National Institute of Standards and Technology, Cryptographic Algorithm Validation Program (2002), https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/aes/AESAVS.pdf
3. Bernstein, D.J.: ChaCha, a variant of Salsa20. In: Workshop Record of SASC 2008: The State of the Art of Stream Ciphers (2008), https://cr.yp.to/chacha.html
4. Coutinho, M., de Oliveira Albuquerque, R., Borges, F., García Villalba, L.J., Kim, T.H.: Learning perfectly secure cryptography to protect communications with adversarial neural cryptography. Sensors **18**(5), 1306 (2018). https://doi.org/10.3390/s18051306
5. CRYPTREC: CRYPTREC ciphers list (e-government recommended ciphers list). CRYPTREC LS-0001-2022R2 (2026), https://www.cryptrec.go.jp/en/list.html
6. Eum, S., Song, M., Kim, S., Seo, H.: Efficient GPU parallel implementation and optimization of ARIA for counter and exhaustive key-search modes. Electronics **14**(10), 2021 (2025). https://doi.org/10.3390/electronics14102021
7. Gérault, D., Hambitzer, A., Ronen, E., Shamir, A.: Deep neural cryptography. In: Advances in Cryptology – EUROCRYPT 2026, Part VI. Lecture Notes in Computer Science, vol. 16546, pp. 513–543. Springer (2026). https://doi.org/10.1007/978-3-032-25333-0_18, full version: Cryptology ePrint Archive, Paper 2025/288
8. Gérault, D., Hambitzer, A., Ronen, E., Shamir, A.: Deep neural cryptography (reference implementation). https://github.com/DavidGerault/deep_neural_cryptography (2026)
9. Gohr, A.: Improving attacks on round-reduced Speck32/64 using deep learning. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology – CRYPTO 2019, Part II. Lecture Notes in Computer Science, vol. 11693, pp. 150–179. Springer (2019). https://doi.org/10.1007/978-3-030-26951-7_6
10. Goldwasser, S., Kim, M.P., Vaikuntanathan, V., Zamir, O.: Planting undetectable backdoors in machine learning models: Extended abstract. In: 63rd Annual Symposium on Foundations of Computer Science (FOCS). pp. 931–942. IEEE (2022). https://doi.org/10.1109/FOCS54457.2022.00092
11. Hochschild, P.H., Turner, P., Mogul, J.C., Govindaraju, R., Ranganathan, P., Culler, D.E., Vahdat, A.: Cores that don't count. In: Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21). pp. 9–16. ACM (2021). https://doi.org/10.1145/3458336.3465297
12. Ignatov, A., Timofte, R., Kulik, A., Yang, S., Wang, K., Baum, F., Wu, M., Xu, L., Van Gool, L.: AI benchmark: All about deep learning on smartphones in 2019. In: 2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW). pp. 3617–3635. IEEE (2019). https://doi.org/10.1109/ICCVW.2019.00447

13. Jin, X., Ma, S., Lin, Z.: Repurposing neural networks for efficient cryptographic computation. In: 32nd Annual Network and Distributed System Security Symposium, NDSS 2025. The Internet Society (2025). https://doi.org/10.14722/ndss.2025.240955, doi: 10.14722/ndss.2025.240955
14. Kim, K.: Assessing geometric security of AES neural realizations: Linear-time key recovery via neural leakage. Cryptology ePrint Archive, Paper 2026/734 (2026), https://eprint.iacr.org/2026/734
15. Lee, J., Lee, J., Kim, J., Kwon, D., Kim, C.: A description of the ARIA encryption algorithm. RFC 5794 (2010). https://doi.org/10.17487/RFC5794
16. Li, J., Xie, W., Li, L., Wu, X.: Parallel implementation and optimization of SM4 based on CUDA. In: Applied Cryptography in Computer and Communications (AC3 2021). LNICST, vol. 386, pp. 93–104. Springer (2021). https://doi.org/10.1007/978-3-030-80851-8_7
17. Li, M., Li, D., Liu, J., Cheung, S.C.: Demystifying the silence of correctness bugs in PyTorch compiler (2026), arXiv:2604.08720
18. Matsui, M., Nakajima, J., Moriai, S.: A description of the Camellia encryption algorithm. RFC 3713 (2004). https://doi.org/10.17487/RFC3713
19. National Institute of Standards and Technology: Advanced encryption standard (AES). Federal Information Processing Standards Publication, FIPS PUB 197 (2001). https://doi.org/10.6028/NIST.FIPS.197-upd1, updated May 9, 2023
20. Ngo, A.T., Chattopadhyay, A., Maitra, S.: Cryptographic backdoor for neural networks: Boon and bane (2025), arXiv:2509.20714
21. Nir, Y., Langley, A.: ChaCha20 and Poly1305 for IETF protocols. RFC 8439 (2018). https://doi.org/10.17487/RFC8439
22. Shen, Q., Ma, H., Chen, J., Tian, Y., Cheung, S.C., Chen, X.: A comprehensive study of deep learning compiler bugs. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21). pp. 968–980. ACM (2021). https://doi.org/10.1145/3468264.3468591
23. Standardization Administration of China: Information security technology — SM4 block cipher algorithm. GB/T 32907-2016 (2016)
24. Yang, P.: ShangMi (SM) cipher suites for TLS 1.3. RFC 8998 (2021). https://doi.org/10.17487/RFC8998