CIPHERFLOW: Compiling AI Programs for Efficient Cryptographically Secure Inference

Hedong Zhang[1], **Mengxin Zheng**[1], **Qian Lou**[1]
[1]LoopLeap Lab
{hedong.zhang,mengxin.zheng,qian.lou}@ucf.edu

Project Page:https://github.com/loopleap-lab/CipherFlow

Abstract

Cryptographically secure inference faces two practical challenges: making secure programming accessible to AI developers and data scientists, and achieving efficient execution across deployment environments. Developers must compose cryptographic primitives correctly, while performance depends on hardware, networks, and evolving ciphertext states. We present CIPHERFLOW, a compiler framework that takes a high-level AI computation graph, available cryptographic primitives, and a target hardware and network configuration to generate an executable, latency-optimized secure inference program. CIPHERFLOW automates cryptographic execution decisions through three techniques: *MicroBench* profiles deployment-specific primitive costs; state-expanded planning jointly optimizes homomorphic encryption (HE) and secure multi-party computation (MPC) placement, conversions, and ciphertext maintenance under cryptographic-state constraints; and *BlockFold* summarizes branched regions to scale planning to Transformer graphs. Evaluation using measured HE/MPC backend costs across CPU/GPU and network settings shows up to 16.5× speedup on BERT-base and over 22× on ViT compared with static hybrid placement baselines. CIPHERFLOW separates model programming from cryptographic execution decisions, enabling developers to express AI computations while the compiler constructs efficient secure programs.

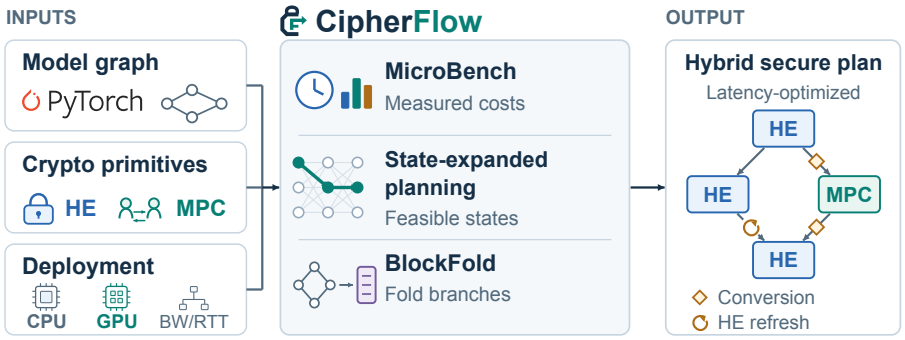


Figure 1: **CipherFlow at a glance.** Model graphs, cryptographic primitives, and hardware/network profiles inform a latency-optimized HE–MPC plan with conversions and HE maintenance.

1 Introduction

AI developers and data scientists increasingly deploy models in privacy-sensitive domains such as healthcare, finance, and biometric authentication, where plaintext cloud inference can expose user

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

inputs and model intellectual property [1–6]. Cryptographically secure inference protects these assets by evaluating neural networks with homomorphic encryption (HE) [7] and secure multi-party computation (MPC) [8], without exposing inputs, model weights, or intermediate activations beyond the prescribed output [9–14]. Turning this capability into a practical way to deploy AI requires addressing two coupled challenges: **programming usability** and **execution efficiency**.

Challenge 1: Programming usability. AI programmers express models through tensor operators and data dependencies; a secure implementation must additionally specify how those computations are realized through cryptographic protocols. The developer must choose HE or MPC for each operator, insert conversions between representations, and maintain compatible ciphertext states along sequential paths and at branch joins [15, 16]. These decisions require cryptographic and systems expertise beyond model development. For example, assigning a matrix multiplication to HE is insufficient if its input ciphertext lacks the required level budget, and a residual addition cannot execute in HE until its inputs have compatible states. Prior secure-computation compilers and mixed-protocol frameworks establish the value of translating high-level programs into cryptographic execution [17, 18]. The programming challenge is therefore to let developers specify the AI computation while a compiler constructs a complete, executable secure program, including the conversions and maintenance actions needed by its backend primitives.

Challenge 2: Execution efficiency. An executable secure program must also adapt to the deployment on which it runs. HE can efficiently process linear operations such as matrix multiplications and convolutions with little interaction [6, 19, 20], whereas nonlinear functions such as ReLU and Softmax can require expensive approximations or ciphertext maintenance [10, 21–26]. MPC offers alternative protocols for these functions but incurs communication and synchronization costs [27, 28]. This complementarity motivates hybrid HE–MPC execution [26, 15, 29]. Figure 2(a,b) illustrates how explicit conversions connect the two domains and how a hybrid strategy can outperform either domain alone. Representative hybrid systems use manually designed placement strategies [30–33], but a fixed assignment by operator type cannot capture changing compute and communication costs.

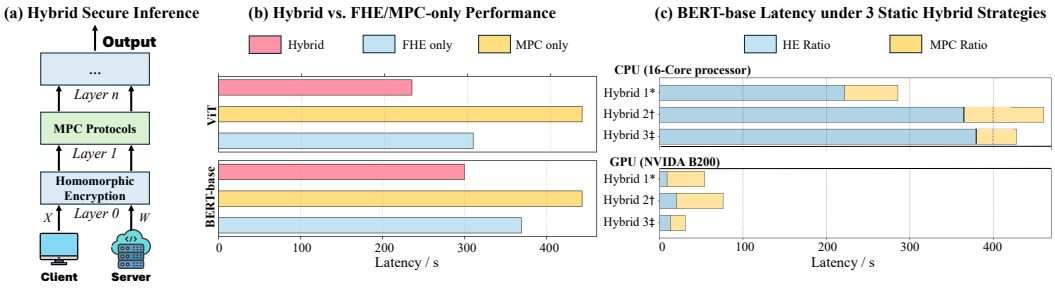


Figure 2: **(a)** Hybrid secure inference executes different operators in HE or MPC with explicit conversions. **(b)** Hybrid execution can outperform FHE-only and MPC-only across different model architectures under the same CPU (16-core) and network setting $200\,\mathrm{Mbps}$, $4\,\mathrm{ms}$. **(c)** BERT-base latency under three static hybrid strategies across different hardware settings. ($*$ Linear$\in$ HE, Nonlinear$\in$ MPC [32]; † Attention$\in$ HE, FFN$\in$ MPC; ‡ Attention$\in$ MPC, FFN$\in$ HE)

Execution efficiency depends on both deployment costs and cryptographic state. GPU acceleration changes the cost of HE relative to MPC [10, 34–41], while bandwidth and round-trip time change the cost of interactive protocols and cross-domain conversions [6, 42–44]. Figure 2(c) shows that the best static strategy for BERT-base changes across CPU and GPU deployments. Moreover, HE operations consume a finite ciphertext level budget [45, 46]; continuing in HE may require bootstrapping [7] or a conversion round trip. At residual and attention joins, incoming states must also be compatible. Consequently, a locally inexpensive operator choice can force costly maintenance or conversions downstream, and enumerating joint states across branches can make planning intractable for Transformer graphs [47]. Efficient compilation must optimize these coupled decisions while enforcing execution feasibility.

CipherFlow: from AI programs to secure execution. We present CIPHERFLOW (Figure 1), a compiler framework that takes a high-level AI computation graph, available cryptographic primitives, and a target hardware and network configuration, and generates a deployment-specialized, latency-optimized secure inference program. It addresses programming usability by separating model computation from cryptographic execution decisions: the compiler selects HE/MPC operators and

materializes conversions, state alignment, and maintenance actions using the supported backends. It addresses execution efficiency by jointly planning these actions under measured deployment costs and cryptographic-state constraints. The resulting program composes the backend primitives under their security assumptions; the compiler automates their selection and execution planning.

CIPHERFLOW combines three techniques. *MicroBench* profiles HE/MPC operators, conversions, and bootstrapping on the target deployment. It benchmarks representative secure operator patterns and reuses measurements across repeated model blocks, keeping profiling costs manageable. *State-expanded planning* represents each operator together with its execution domain and remaining HE level, and jointly optimizes placement, conversion, and maintenance decisions. This turns feasibility-aware planning on sequential regions into shortest-path search. *BlockFold* extends planning to branched graphs by summarizing residual and attention regions as boundary-state transition tables. These summaries capture feasible, minimum-cost transformations through each region and support their composition without enumerating all branch-internal state combinations across the full graph. Together, the three techniques connect a high-level programming interface to executable secure computation tailored to its deployment.

We make four contributions:

1. **A compiler framework for usable and efficient secure AI inference.** We formulate compilation from AI computation graphs, cryptographic primitives, and deployment configurations to executable hybrid secure programs, jointly capturing operator placement, conversions, ciphertext maintenance, and cryptographic-state feasibility.
2. **Deployment-specific cost profiling and state-aware planning.** We design *MicroBench* and a state-expanded graph formulation that jointly optimize HE/MPC execution and auxiliary actions under measured hardware and network costs, reducing sequential planning to shortest-path search.
3. **Scalable planning for branched neural graphs.** We introduce *BlockFold*, which summarizes branched regions using boundary-state transition tables and composes them into a global plan while respecting compatibility at joins, enabling optimization on Transformer-scale graphs.
4. **Implementation and evaluation with real cryptographic backends.** We implement CIPHERFLOW on HE/MPC backends and evaluate plans using measured operator latency, communication, and maintenance costs across CPU/GPU and network settings. On BERT-base and ViT, the evaluation shows up to 16.5× and over 22× speedup, respectively, compared with static hybrid placement baselines [32, 33, 42].

2 Preliminaries & Problem Formulation

2.1 Preliminaries

We summarize the primitives and cost quantities used by the compiler.

Secure Multi-Party Computation. We use semi-honest two-party computation (2PC) over additive secret shares [17, 48]: a value x is split as $[x]_0 + [x]_1 \equiv x \pmod{2^\ell}$. Linear operations are evaluated locally, while nonlinear operations invoke interactive sub-protocols [28, 49]. Thus, MPC latency depends on local compute, communication volume b, and round count r under bandwidth BW and round-trip time (RTT).

Homomorphic Encryption. We use a leveled CKKS-style scheme [10, 46]. Each ciphertext carries a finite level budget L; executing operator v_i in HE consumes δ_i levels, and if the remaining level $l < \delta_i$, a costly bootstrap is required to refresh the ciphertext [7]. HE additions at residual or attention joins further require compatible ciphertext levels.

HE-MPC Conversion. Hybrid execution moves intermediate values across the two domains via HE→MPC and MPC→HE. Both carry measurable compute and communication cost. An HE→MPC→HE round trip restores a fresh HE state and can therefore substitute for a bootstrap, but it is not free: it pays two conversions, whose communication-dominated cost trades off against the compute-bound cost of a direct bootstrap depending on the deployment (Section 3.2).

Compilers and cost models for secure computation. Prior compilers and mixed-protocol frameworks, including EzPC and MP2ML, translate high-level programs into secure-computation protocols [17, 18], while HE compilers such as nGraph-HE2 optimize neural-network inference within a single HE backend [50–52]. Cost-modeling systems such as COSTCO show that secure-computation performance depends strongly on operator shape, hardware, and network conditions [43]. Our compiler is complementary but targets a different optimization problem: it jointly plans HE/MPC placement, bidirectional conversions, and HE level maintenance over branched DNN graphs.

Private neural-network and Transformer inference optimizations. Other work improves private inference by optimizing the underlying protocols, approximations, or model structures [12, 29, 30, 53, 54]. Cheetah reduces communication in two-party neural-network inference [6]; MPCFormer and related systems optimize MPC-based Transformer inference [55]; HE-oriented systems such as THOR and Orion improve encrypted Transformer or neural-network execution through packing, rotation, and polynomial-evaluation optimizations [22, 23]; and PrivCirNet exploits structured model transformations to reduce private-inference cost [56]. These approaches improve specific backends or model designs, whereas our contribution is a hardware-aware compiler that uses measured primitive costs to choose a low-latency hybrid execution plan for each deployment. Closest to our setting, CostSphere [57] assigns FHE/MPC protocols with an ILP over a linearized operator sequence using learned CPU cost models that adapt to network conditions; it approximates HE state by a consecutive-FHE count and does not model join compatibility or hardware changes. Matchmaker [58] splits batches between LSS- and FSS-based MPC on CPU and GPU to maximize throughput, but stays within MPC and does not place individual operators. EncFormer [59] improves the hybrid stack with stage-compatible FHE kernels and a complex CKKS–MPC conversion under hand-designed FHE/MPC boundaries. Their protocols and kernels can be profiled by MicroBench as additional actions; CipherFlow contributes deployment-specialized planning of placement, conversion, and ciphertext-state maintenance over branched graphs (Appendix A).

2.2 Threat Model

We target standard cloud secure inference: the client holds private input X, the server holds model weights W, and only the agreed output is revealed. We assume a semi-honest two-party adversary, as widely adopted in secure inference [5, 6, 9, 10, 14, 28, 32, 33, 42, 60, 61]. Parties follow the protocol but may infer extra information from their local views; the client keeps key material locally, and the two parties do not collude with any external party. Malicious security, side channels, integrity, and verifiability are orthogonal to our compiler focus and can be incorporated through complementary techniques [23, 42].

2.3 Problem Formulation

We formalize hardware-aware hybrid compilation as a constrained latency-minimization problem. The input consists of: *(i)* a user model represented as a computation graph $G = (V, E)$, where $V = \{v_1, \ldots, v_N\}$ denotes the operators and E their data dependencies; and *(ii)* a deployment-specific cost profile $\mathcal{D}$ produced by MicroBench module under target hardware $\mathcal{H}$ and network conditions $(\mathrm{BW}, \mathrm{RTT})$, which records the profiled compute, communication, and maintenance costs of HE/MPC operators, cross-domain conversions, and bootstrapping.

The output is a hybrid secure execution plan P that assigns each operator $v_i \in V$ a cryptographic execution state $S_i = (d_i, l_i)$, where $d_i \in \{\mathrm{HE}, \mathrm{MPC}\}$ is the execution domain and, when $d_i = \mathrm{HE}$, l_i denotes the remaining HE level. In addition to operator states, the plan specifies the auxiliary actions required to realize valid transitions across the graph, including HE↔MPC conversions and HE maintenance actions such as bootstrapping.

A plan is valid if it preserves the functional semantics of G and if its induced state assignments and transitions are feasible under the underlying cryptographic primitives. In particular, HE-domain operators must respect the level budget $l_i \geq 0$, and HE merge points must satisfy operand-compatibility constraints across incoming branches.

The objective is to find, among all valid plans, the one of minimum end-to-end inference latency:

$$P^* \;=\; \arg \min_{P \in \mathcal{P}(G,\mathcal{D})} \;\; \mathrm{Latency}_{\mathrm{e2e}}(G, P; \mathcal{D}), \tag{1}$$

where $\mathcal{P}(G, \mathcal{D})$ denotes the set of valid plans for graph G under deployment setting $\mathcal{D}$, and $\mathrm{Latency}_{\mathrm{e2e}}$ is computed by composing the per-state and per-transition costs profiled by *MicroBench*. The central technical problem is to solve Eq. (1) efficiently on DNN graphs, where decisions are coupled by HE state evolution along paths and compatibility constraints across branches.

3 CipherFlow Compiler Framework

CIPHERFLOW addresses the two challenges in Section 1 through a single compilation pipeline: it automates cryptographic execution decisions for *programming usability* and jointly optimizes their deployment-specific costs for *execution efficiency*. Figure 3 shows how a high-level AI computation becomes an executable hybrid HE–MPC program.

Compiler inputs and output. The inputs are a model graph $G = (V, E)$, the available HE/MPC primitive implementations and conversion routines, and a target hardware and network configuration $(\mathcal{H}, \mathrm{BW}, \mathrm{RTT})$. The primitives determine the supported execution actions and their state transitions; *MicroBench* supplies the deployment-specific cost profile $\mathcal{D}$ used in Eq. (1). The output is a secure execution plan P that specifies backend operators, intermediate representations, HE↔MPC conversions, and ciphertext alignment and maintenance actions. These decisions make the plan executable using the supported primitives, beyond merely labeling operators as HE or MPC.

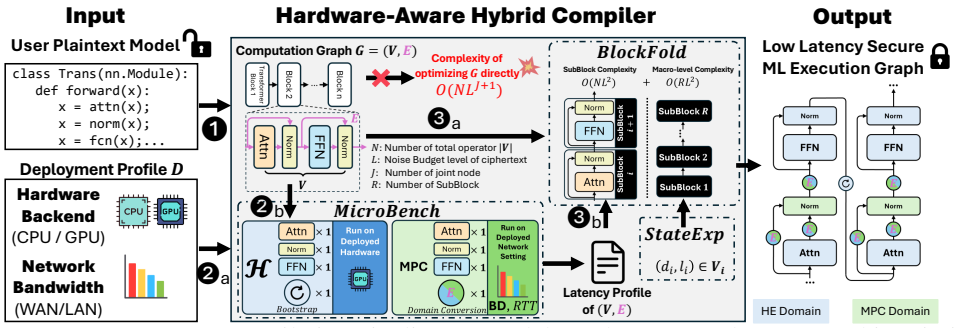


Figure 3: CIPHERFLOW compilation pipeline. A model graph, supported cryptographic primitives, and deployment information determine a secure execution plan. *MicroBench* supplies measured costs; state-expanded planning and *BlockFold* select feasible operator, conversion, and maintenance actions. Circled numbers denote pipeline *stages*: boxes sharing a number (e.g., ②a/②b) are independent components whose outputs jointly feed the next stage.

Addressing programming usability. AI programmers specify tensor computations and dependencies while the compiler determines their cryptographic realization. State-expanded planning tracks representations and HE level budgets, and *BlockFold* resolves compatibility across branches. The selected plan explicitly includes conversions, alignment, and refresh actions, so programmers do not manually schedule these operations. The compiler plans within the supported primitives' execution constraints and security assumptions; state feasibility is not a replacement for the underlying protocols' security guarantees.

Addressing execution efficiency. *MicroBench* measures the costs of available actions on the target deployment. State-expanded planning compares complete feasible sequences, jointly choosing operator placement, conversion points, and HE maintenance rather than selecting the cheapest operator in isolation. *BlockFold* makes this optimization tractable on branched graphs by composing region summaries. Thus, the same model can be replanned for different hardware or networks without requiring the programmer to rewrite its cryptographic execution strategy.

3.1 MicroBench: Deployment-Specific Cost Instantiation

MICROBENCH addresses execution efficiency by grounding the optimizer in deployment-specific measurements. It runs at compilation time, rather than per inference, and profiles HE/MPC execution, cross-domain conversions, and bootstrapping. This supplies the cost model automatically instead of requiring programmers to estimate which cryptographic primitive will be fastest.

The model is lowered to an operator-level DAG, from which MICROBENCH extracts distinct operator-shape patterns and profiles the available backends and conversion routines. Repeated tensor shapes

and block structures in Transformer models allow measurements to be reused across operator instances. For each action, MICROBENCH records local compute time, communication volume, and round count when applicable. The planner composes these quantities with the target bandwidth and RTT to obtain the costs of execution, HE↔MPC conversion, and bootstrapping. The resulting profile $\mathcal{D}$ assigns costs to the legal actions used by the planner; the primitives' state-transition rules determine their feasibility. Additional profiling details are given in Appendix B.

3.2 State-Expanded Planning

State-expanded planning connects programming usability to execution efficiency. It represents cryptographic states explicitly, so the compiler can insert the actions required for execution and compare their combined costs. An HE operator's feasibility depends on the inherited ciphertext state, especially its remaining level; an inexpensive placement can therefore cause expensive refreshes or conversions downstream. Planning over a *state-expanded graph* captures these dependencies without asking the programmer to manage ciphertext levels manually.

State-expanded nodes. A node is a triple $s = (i, d, l)$, pairing an operator index $i \in \{0, \ldots, N+1\}$ (with sentinel source and sink) with the state $S_i = (d, l)$ from Section 2.3. Here $d \in \{\mathrm{HE}, \mathrm{MPC}\}$ is the execution domain and l is the remaining HE level. For MPC, the level component is inactive; we write $l = L$ by convention, consistent with an MPC→HE conversion returning a fresh level budget. Including the inherited state in each node makes the residual planning cost locally additive along execution paths.

Legal actions as state transitions. Edges represent five compiler actions: HE execution, MPC execution, HE→MPC conversion, MPC→HE conversion, and bootstrapping. Execution advances the operator index, conversion switches domains without consuming an operator, and bootstrapping refreshes the HE state. Only transitions satisfying the supported actions' state requirements are admissible. Edge weights come from MICROBENCH's deployment-specific profile; the full transition rules are listed in Appendix E. Selecting a path therefore specifies both operator execution and the auxiliary protocol actions needed to make that execution feasible.

Refresh by bootstrapping or by conversion. Both a direct bootstrap and an HE→MPC→HE round trip restore the ciphertext level. Bootstrapping is compute-only, whereas the round trip is communication-dominated, so their relative cost depends on the deployment. The planner chooses between these alternatives together with operator placement, relieving the programmer of manually selecting refresh points and mechanisms. Table 3 illustrates how the preferred strategy changes with hardware and bandwidth.

Sequential shortest-path reduction. On a sequential computation chain, a valid hybrid plan corresponds to a path from source state s_{src} to sink state s_{dst}. Its modeled latency is the sum of the edge costs, so the minimum-cost path gives the optimal sequential plan under the profiled cost model. This jointly determines operator placement, conversions, and HE maintenance. The state-expanded graph contains $O(NL)$ nodes and edges on a chain, keeping planning tractable for the HE level ranges considered. Branched graphs additionally require compatibility across incoming paths; *BlockFold* extends this formulation to those dependencies.

3.3 BlockFold: Hierarchical Planning for Branched Graphs

BlockFold preserves the same programming abstraction for residual and attention branches: the input graph specifies their dependencies, and the compiler determines compatible cryptographic execution states at each join. Tracking every combination of branch states globally would make planning grow combinatorially. *BlockFold* instead summarizes single-entry single-exit (SESE) regions and composes their boundary-state effects, supporting efficient planning while retaining the state constraints needed for execution.

Step 1: SESE region extraction. The compiler identifies subgraphs with a unique entry point and a unique exit point, potentially containing internal branches and merges. Residual blocks, self-attention blocks, and multi-head attention structures provide such regions. Isolating them separates branch-local state interactions from the rest of the graph without requiring the programmer to annotate their HE/MPC placement.

Step 2: Region transfer functions. For each SESE region R, *BlockFold* solves a local planning problem over feasible execution, conversion, and maintenance actions. It enforces operand compatibility at HE merges (Section 2.1) through level-dropping or targeted bootstrapping. The region transfer function

$$T_R(s^R_{\mathrm{src}}, s^R_{\mathrm{dst}}) = \min_{P:\, s^R_{\mathrm{src}} \rightsquigarrow s^R_{\mathrm{dst}}} \sum_{e \in P} c(e)$$

records the minimum cost of traversing R from input state s^R_{src} to output state s^R_{dst}. Each entry summarizes feasible internal choices of HE/MPC placement, conversions, level alignment, and bootstrapping, exposing their boundary-state effect to the global planner.

Step 3: Macro-level composition. Replacing each region with its transfer function yields an aggregate graph over region boundaries. A macro-level shortest-path search composes these summaries into a globally coordinated plan. This avoids tracking all branch-internal joint states across the full graph. The complexity analysis appears in Appendix F, and the full algorithm is given in Appendices D–E.

From an optimized plan to executable secure computation. After selecting the macro path, a backtracking pass expands the recorded region solutions onto the original graph. It materializes operator placement, cross-domain conversions, level alignment, and bootstrapping while retaining the graph's dependencies. The resulting plan P specifies the backend operations and auxiliary actions required for execution. This completes the connection between the two challenges: the compiler takes over cryptographic execution decisions from the AI programmer, while measured costs and state-aware optimization determine an efficient realization for the target deployment.

4 Evaluation

We organize the evaluation around *programming usability* and *execution efficiency*. For programming usability, we examine HE-state feasibility and the scalability of automated plan generation. For execution efficiency, we examine deployment-specific inference latency and the contribution of cost modeling and joint planning. The usability evidence concerns compiler functionality and compilation cost; programmer effort is not measured. We first describe the experimental setup, then report execution latency, and finally group the component ablations by these two goals.

4.1 Experimental Setup

Hardware. We evaluate on two target platforms: **CPU Platform:** A workstation equipped with an AMD Ryzen Threadripper PRO 3955WX 16-Core processor. **GPU Platform:** An HPC cluster equipped with an NVIDIA B200 GPU.

Network and workload. We sweep bandwidth $\mathrm{BW} \in \{10\ \mathrm{Mbps}, 100\ \mathrm{Mbps}, 1\ \mathrm{Gbps}, 3\ \mathrm{Gbps}\}$ and $\mathrm{RTT} = \{1, 20, 40, 80\}\ \mathrm{ms}$, yielding 16 operating points per platform spanning constrained WAN to high-bandwidth LAN. We evaluate two workloads: BERT-base and ViT-B/16. BERT-base uses 12 Transformer blocks at sequence length 128, including attention QK/AV, out-projection, FFN-Linear-1/2, Softmax, LayerNorm, GeLU, and residual additions; the compiler plans over 132 operator instances. ViT-B/16 uses 197 tokens (14×14 patches plus a CLS token) with repeated self-attention, MLP blocks, LayerNorm, and residual connections, allowing us to evaluate whether the compiler generalizes beyond language Transformer inference.

Primitives. HE primitives are NEXUS CKKS on CPU and NEXUS-CUDA ($N = 32768$) on GPU, both with a 20-level budget and bootstrapping. MPC primitives use SCI/SHAFT [5, 14, 62] on CPU and SHAFT [14] on GPU. HE and MPC operator latencies are measured as wall-clock time on the target hardware (Table 14); SHAFT's bytes and rounds are its measured end-to-end totals apportioned to operators by compute time. HE↔MPC conversion is profiled by MicroBench under standard mask-and-decrypt and re-encryption protocols. We provide additional details on MicroBench, including its warm-up protocol, profiling procedureand representative operator-level cost entries, in Appendix I.

Baselines. BumbleBee, BOLT, and BLB are applied as operator-to-domain placement rules (linear operators in HE, nonlinear operators in MPC) on our primitives; BOLT and BLB additionally receive a discount on conversion cost ($0.75\times$ and $0.5\times$) that models their batched and fused conversions. For BERT-base we also include the published BumbleBee end-to-end system measurement [33], recomposed under each network setting, as a stronger system-level baseline. All models use static shapes (fixed sequence length); Section 5 discusses dynamic shapes.

4.2 Execution Efficiency Across Deployments

Figure 4 (a,b) reports end-to-end BERT-base latency across four bandwidth settings on the CPU and GPU. Our compiler consistently matches or outperforms the best static placement baseline at every operating point. Apart from the published BumbleBee system measurement (orange bars in (a,b)), all bars within each group use the same profiled HE, MPC, conversion, and maintenance costs, so these improvements isolate the benefit of placement and conversion planning rather than differences in primitive implementations.

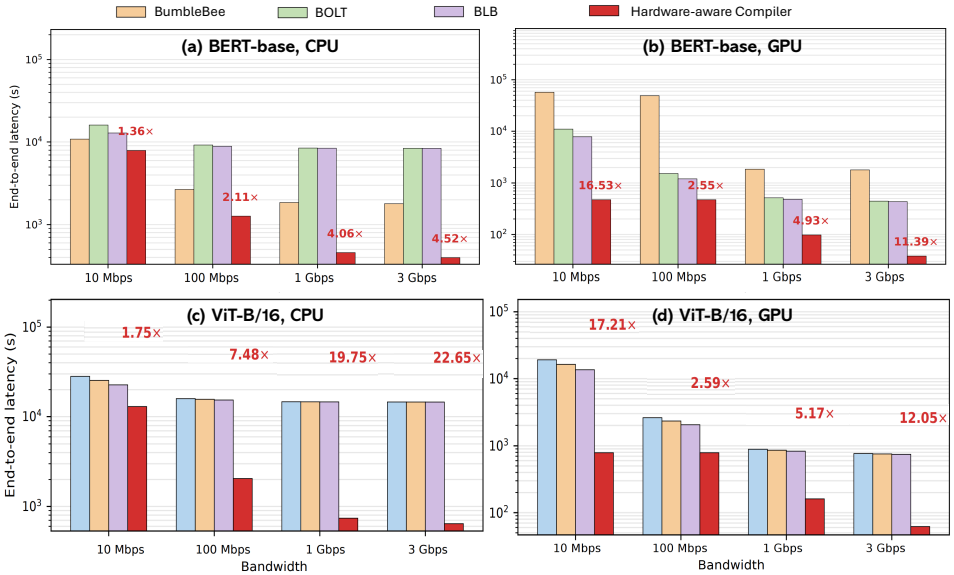


Figure 4: End-to-end latency of CIPHERFLOW versus static hybrid baselines on BERT-base (a,b) and ViT-B/16 (c,d). For BERT-base, the orange bars are the published BumbleBee end-to-end system measurement recomposed under each network; BOLT and BLB (and all ViT baselines) apply the corresponding placement rules to our primitives. Panel (a) reports RTT $= 1$ ms, where the BumbleBee system is strongest; panels (b–d) report the RTT with the largest gap for each bandwidth. Speedup is relative to the fastest baseline in each group. The full 4×4 sweep, including placement-fair BumbleBee mappings, is in Appendix H.

The key observation is that the best static strategy changes with the deployment setting. On CPU, the relative ranking of static baselines varies as bandwidth increases, showing that a fixed HE/MPC assignment cannot robustly capture the trade-off between local cryptographic computation and communication. On GPU, the latency pattern differs from CPU, indicating that hardware acceleration shifts the HE-MPC cost balance and changes which placement is preferable. Therefore, the optimal BERT-base plan is both bandwidth-dependent and hardware-dependent. Appendix G further visualizes the compiler-selected HE/MPC operator mix across bandwidth and RTT settings, showing how the same BERT-base graph induces different placements under CPU and GPU primitive profiles.

These results support the need for a compiler rather than a manually designed hybrid rule. A static strategy may perform well in one regime but become suboptimal when moved to another bandwidth or hardware setting. In contrast, our compiler uses the deployment-specific profile to select the lowest-latency plan for each operating point, explaining why the red bars remain at or below the best static baseline across Figure 4.

Figure 4 (c,d) shows that the compiler provides even larger gains on ViT, where static hybrid mappings suffer from the longer token sequence. ViT-Base/16 uses $S = 197$ tokens, compared with $S = 128$ for BERT-base. This increases attention QK/AV costs quadratically with S and FFN and conversion costs linearly with S. Static linHE+nlMPC-style mappings therefore pay amplified HE matmul latency while still inserting HE↔MPC conversions around nonlinear operators. In contrast, the compiler re-optimizes the entire placement under the measured deployment profile: when bandwidth is sufficiently high, it can move ViT to an all-MPC plan, eliminating conversion overhead and avoiding the enlarged HE attention bottleneck. BERT-base has a shorter sequence length, so HE

matmuls remain more competitive and the compiler often selects a mixed plan, leading to smaller but still consistent gains.

4.3 Component Analysis: Usability and Efficiency

The ablations connect the compiler components to execution efficiency through plan quality, and to programming usability through feasible and scalable automated planning. Table 1 reports the worst-case end-to-end latency ratio of each ablated planner relative to the full compiler across the evaluated BERT-base deployment settings. Table 2 reports the corresponding compilation cost on BERT-base.

4.3.1 Execution Efficiency: Cost Modeling and Plan Quality

Table 1: Plan-quality ablation and per-operator baselines.

Ablation	CPU latency ratio	GPU latency ratio
Full	1.00×	1.00×
−MicroBench	1.19×	19.14×
−StateExp	1.00×	1.00×
−BlockFold	1.00×	1.00×
−Bootstrap	1.00×	3.88×
Greedy per-operator	1.37×	5.75×
Random, best of 100	10.92×	14.02×

MicroBench: deployment-specific cost modeling. Removing MicroBench replaces measured deployment-specific primitive costs with a fixed cost model. This directly degrades plan quality because the planner can no longer observe how the relative costs of HE execution, MPC execution, conversion, and maintenance change across CPU/GPU backends and network settings. As shown in Table 1, this ablation increases the worst-case latency by 1.19× on CPU and by 19.14× on GPU. The much larger degradation on GPU confirms the central motivation of CIPHERFLOW: hardware acceleration can substantially change the optimal hybrid strategy, so a static or deployment-agnostic cost model is insufficient.

Per-operator baselines. Greedy assigns each operator to the domain with lower local cost, followed by the necessary conversions and bootstraps. Random samples 100 uniform per-operator assignments. We evaluate all plans using the same cost model. Greedy performs no better than CipherFlow, particularly when adjacent operators favor different domains. On GPU at 10 Mbps, Greedy assigns 24 of 132 operators to MPC, requiring 72 conversions and resulting in a $5.75\times$ slowdown compared with CipherFlow's all-HE plan. Even the best random assignments are 10.9–$14.0\times$ slower (Table 13). EncFormer's placement [59], which assigns linear operators to FHE and nonlinear operators to MPC with half-payload conversions, matches the BLB column under our profiles.

4.3.2 Programming Usability: Feasibility and Scalability

Table 2: Compilation-time ablation on BERT-base.

Ablation	Compilation time	Ratio vs. Full	Method
Full	3.50 s	1.00×	empirical
−MicroBench	3.39 s	0.96×	empirical
−StateExp	2.36 s	0.67×	empirical, infeasible
−BlockFold	>278 h	$> 2.8 \times 10^5\times$	search-space extrapolation

State-expanded planning: HE-state feasibility. Removing state expansion makes the planner optimize only over operator domains, without explicitly tracking the remaining HE level along each execution path. This does not necessarily increase the repaired end-to-end latency in Table 1, but the apparent result is misleading: without state expansion, the planner can select plans that violate HE feasibility constraints and therefore require after-the-fact repair. In contrast, the full compiler reasons

over execution domain and HE state jointly, so placement, conversion, and maintenance decisions are optimized under feasibility constraints rather than patched after planning. The reduced compile time of the −StateExp variant in Table 2 therefore reflects a weaker and potentially infeasible optimization problem, not a better compiler.

BlockFold: scalable optimization over branched graphs. Removing BlockFold preserves the same local state-expanded formulation but applies it directly to the full branched graph. Its worst-case latency in Table 1 is unchanged, because BlockFold preserves the optimal boundary-state cost of each folded region; the chain relaxation used when BlockFold is removed can even report up to 11% lower latency on GPU, but only by ignoring join-compatibility constraints. Its benefit is scalability. As shown in Table 2, the full compiler produces a BERT-base plan in 3.51 seconds, whereas the planner without BlockFold does not finish within the search budget and is extrapolated to require more than 278 hours, corresponding to a slowdown greater than $2.8 \times 10^5\times$. With BlockFold, compile time grows linearly with depth: 3.5, 7.3, 14.8, and 29.9 s for 12, 24, 48, and 96 encoder blocks. This result shows that BlockFold is necessary for extending HE-state-aware planning from sequential operator chains to Transformer-scale graphs with residual and attention joins.

The ablations connect the two goals to complementary compiler components. For execution efficiency, MicroBench provides the measured deployment-specific costs needed for low-latency plan quality. For programming usability, state-expanded planning enforces HE-state feasibility during optimization, and BlockFold makes automated state-aware planning scalable on branched neural-network graphs.

5 Discussion

Programming usability through compilation. CipherFlow separates model development from cryptographic execution planning. AI programmers specify a model graph; the compiler combines available HE/MPC primitives with the target hardware and network configuration to construct an operator-level secure execution plan. State-expanded planning makes conversions, HE maintenance, and state compatibility explicit, while BlockFold makes planning scalable across branched graphs. The evaluation examines this compiler-level usability through feasibility and compilation cost; programmer effort is not measured. Security relies on the supported primitives and their assumptions.

Execution efficiency through deployment-specific planning. Our results suggest that HE–MPC placement is better handled as a deployment-specific compilation problem than through fixed operator-mapping rules. Using measured primitive costs from MicroBench, CipherFlow jointly selects operator domains, conversions, and HE maintenance operations to adapt to different hardware and network conditions. Thus, the same abstraction that automates cryptographic decisions also allows their combined execution cost to be optimized for the target deployment.

Broader objectives and batching. Although we optimize latency, other additive objectives, such as monetary cost or weighted latency and communication volume, can be incorporated by changing edge weights. Hard communication budgets require tracking cumulative volume in the planning state. Batching may also change the preferred placement: CKKS SIMD packing can amortize HE computation across requests, while MPC communication scales with the processed data. Batch-aware planning can use profiled batch costs, but jointly optimizing packing and placement, as explored for packing in Coyote [63], remains open.

An extensible platform for private inference. CipherFlow is open source (github.com/loopleap-lab/CipherFlow). New HE/MPC libraries and kernels can be integrated through registered methods and measured costs, allowing the planner to benefit from advances in either backend while retaining the model-level programming interface. Future work includes generative models and encrypted execution.

6 Acknowledgment

This work was supported in part by the National Science Foundation (NSF) under Award Nos. 2523407 and 2413232. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Yancheng Zhang, Mengxin Zheng, Xun Chen, Jingtong Hu, Weidong Shi, Lei Ju, Yan Solihin, and Qian Lou. zkvc: Fast zero-knowledge proof for private and verifiable computing. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, pages 1–7. IEEE, 2025.

[2] Fahad Shamshad, Salman Khan, Syed Waqas Zamir, Muhammad Haris Khan, Munawar Hayat, Fahad Shahbaz Khan, and Huazhu Fu. Transformers in medical imaging: A survey. *Medical image analysis*, 88: 102802, 2023.

[3] Peng Cheng and Utz Roedig. Personal voice assistant security and privacy—a survey. *Proceedings of the IEEE*, 110(4):476–507, 2022.

[4] Prodip Kumar Sarker, Qingjie Zhao, and Md Kamal Uddin. Transformer-based person re-identification: a comprehensive review. *IEEE Transactions on Intelligent Vehicles*, 9(7):5222–5239, 2024.

[5] Deevashwer Rathee, Mayank Rathee, Nishant Kumar, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. Cryptflow2: Practical 2-party secure inference. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 325–342, 2020.

[6] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure {Two-Party} deep neural network inference. In *31st USENIX security symposium (USENIX Security 22)*, pages 809–826, 2022.

[7] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[8] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th annual symposium on foundations of computer science (Sfcs 1986)*, pages 162–167. IEEE, 1986.

[9] Ruisi Zhang, Yifei Zhao, Neusha Javidnia, Mengxin Zheng, and Farinaz Koushanfar. Attestllm: Efficient attestation framework for billion-scale on-device llms. *arXiv preprint arXiv:2509.06326*, 2025.

[10] Jiawen Zhang, Xinpeng Yang, Lipeng He, Kejia Chen, Wen-jie Lu, Yinghao Wang, Xiaoyang Hou, Jian Liu, Kui Ren, and Xiaohu Yang. Secure transformer inference made non-interactive. *Cryptology ePrint Archive*, 2024.

[11] Jiaqi Xue, Yifei Zhao, Mengxin Zheng, Fan Yao, Yan Solihin, and Qian Lou. Securing transformer-based ai execution via unified tees and crypto-protected accelerators. *arXiv preprint arXiv:2507.03278*, 2025.

[12] Qian Lou, Yilin Shen, Hongxia Jin, and Lei Jiang. Safenet: A secure, accurate and fast neural network inference. In *International Conference on Learning Representations*, 2021.

[13] Runhua Xu, Nathalie Baracaldo, and James Joshi. Privacy-preserving machine learning: Methods, challenges and directions. *arXiv preprint arXiv:2108.04417*, 2021.

[14] Andes YL Kei and Sherman SM Chow. Shaft: Secure, handy, accurate, and fast transformer inference. *Cryptology ePrint Archive*, 2025.

[15] Qian Lou, Song Bian, and Lei Jiang. Autoprivacy: Automated layer-wise parameter selection for secure neural network inference. *Advances in Neural Information Processing Systems*, 33:8638–8647, 2020.

[16] Qian Lou and Lei Jiang. Hemet: A homomorphic-encryption-friendly privacy-preserving mobile neural network architecture. In *International conference on machine learning*, pages 7102–7110. PMLR, 2021.

[17] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: Programmable and efficient secure two-party computation for machine learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 496–511. IEEE, 2019.

[18] Fabian Boemer, Rosario Cammarota, Daniel Demmler, Thomas Schneider, and Hossein Yalame. Mp2ml: A mixed-protocol machine learning framework for private inference. In *Proceedings of the 15th international conference on availability, reliability and security*, pages 1–10, 2020.

[19] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th USENIX security symposium (USENIX security 18)*, pages 1651–1669, 2018.

[20] Erwin Eko Wahyudi, Yan Solihin, and Qian Lou. Efficient arithmetic-and-comparison homomorphic encryption with space switching. In *2026 IEEE Symposium on Security and Privacy (SP)*, pages 3168–3184. IEEE, 2026.

[21] Qian Lou and Lei Jiang. She: A fast and accurate deep neural network for encrypted data. *Advances in neural information processing systems*, 32, 2019.

[22] Austin Ebel, Karthik Garimella, and Brandon Reagen. Orion: A fully homomorphic encryption framework for deep learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 734–749, 2025.

[23] Jungho Moon, Dongwoo Yoo, Xiaoqian Jiang, and Miran Kim. THOR: Secure transformer inference with homomorphic encryption. Cryptology ePrint Archive, Paper 2024/1881, 2024. URL `https://eprint.iacr.org/2024/1881`.

[24] Yancheng Zhang, Xun Chen, and Qian Lou. Hebridge: Connecting arithmetic and logic operations in fv-style he schemes. In *Proceedings of the 12th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 23–35, 2023.

[25] Ardhi Wiratama Baskara Yudha, Jiaqi Xue, Qian Lou, Huiyang Zhou, and Yan Solihin. Boostcom: Towards efficient universal fully homomorphic encryption by boosting the word-wise comparisons. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*, pages 121–132, 2024.

[26] Jiaqi Xue, Xin Xin, Wei Zhang, Mengxin Zheng, Qianqian Song, Minxuan Zhou, Yushun Dong, Dongjie Wang, Xun Chen, Jiafeng Xie, et al. Sok: Can fully homomorphic encryption support general ai computation? a functional and cost analysis. *arXiv preprint arXiv:2504.11604*, 2025.

[27] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia conference on computer and communications security*, pages 707–721, 2018.

[28] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. *Advances in Neural Information Processing Systems*, 34:4961–4973, 2021.

[29] Yancheng Zhang, Jiaqi Xue, Mengxin Zheng, Mimi Xie, Mingzhe Zhang, Lei Jiang, and Qian Lou. Cipherprune: Efficient and scalable private transformer inference. In *International Conference on Learning Representations*, volume 2025, pages 52967–52985, 2025.

[30] Mengxin Zheng, Qian Lou, and Lei Jiang. Primer: Fast private transformer inference on encrypted data. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[31] Meng Hao, Hongwei Li, Hanxiao Chen, Pengzhi Xing, Guowen Xu, and Tianwei Zhang. Iron: Private inference on transformers. *Advances in neural information processing systems*, 35:15718–15731, 2022.

[32] Tianshi Xu, Wen-jie Lu, Jiangrui Yu, Yi Chen, Chenqi Lin, Runsheng Wang, and Meng Li. Breaking the layer barrier: Remodeling private transformer inference with hybrid {CKKS} and {MPC}. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 2653–2672, 2025.

[33] Wen-jie Lu, Zhicong Huang, Zhen Gu, Jingyu Li, Jian Liu, Cheng Hong, Kui Ren, Tao Wei, and WenGuang Chen. Bumblebee: Secure two-party inference framework for large transformers. *Cryptology ePrint Archive*, 2023.

[34] Hao Yang, Shiyu Shen, Wangchen Dai, Lu Zhou, Zhe Liu, and Yunlei Zhao. Phantom: A cuda-accelerated word-wise homomorphic encryption library. *IEEE Transactions on Dependable and Secure Computing*, 21 (5):4895–4906, 2024.

[35] Lei Jiang, Qian Lou, and Nrushad Joshi. Matcha: A fast and energy-efficient accelerator for fully homomorphic encryption over the torus. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 235–240, 2022.

[36] Mengxin Zheng, Fan Chen, Lei Jiang, and Qian Lou. Priml: An electro-optical accelerator for private machine learning on encrypted data. In *2023 24th International Symposium on Quality Electronic Design (ISQED)*, pages 1–7. IEEE, 2023.

[37] Xianglong Deng, Shengyu Fan, Zhicheng Hu, Zhuoyu Tian, Zihao Yang, Jiangrui Yu, Dingyuan Cao, Dan Meng, Rui Hou, Meng Li, et al. Trinity: A general purpose fhe accelerator. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 338–351. IEEE, 2024.

[38] Mengxin Zheng, Cheng Chu, Qian Lou, Nathan Youngblood, Mo Li, Sajjad Moazeni, and Lei Jiang. Ofhe: an electro-optical accelerator for discretized tfhe. In *Proceedings of the 29th ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 1–6, 2024.

[39] Mingqin Han, Yilan Zhu, Qian Lou, Zimeng Zhou, Shanqing Guo, and Lei Ju. coxhe: A software-hardware co-design framework for fpga acceleration of homomorphic computation. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1353–1358. IEEE, 2022.

[40] Fan Li, Mayank Kumar, Ruizhi Zhu, Mengxin Zheng, Qian Lou, and Xin Xin. Reliafhe: Resilient design for fully homomorphic encryption accelerators. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1605–1621, 2026.

[41] Mengxin Zheng, Qian Lou, Fan Chen, Lei Jiang, and Yongxin Zhu. Cryptolight: An electro-optical accelerator for fully homomorphic encryption. In *Proceedings of the 17th ACM International Symposium on Nanoscale Architectures*, pages 1–2, 2022.

[42] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. Bolt: Privacy-preserving, accurate and efficient inference for transformers. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4753–4771. IEEE, 2024.

[43] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. Costco: An automatic cost modeling framework for secure multi-party computation. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 140–153. IEEE, 2022.

[44] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A {GPU} platform for secure computation. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 827–844, 2022.

[45] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.

[46] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International conference on the theory and application of cryptology and information security*, pages 409–437. Springer, 2017.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[48] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[49] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. Sirnn: A math library for secure rnn inference. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1003–1020. IEEE, 2021.

[50] Mayank Kumar, Jiaqi Xue, Mengxin Zheng, and Qian Lou. Fhe-coder: Benchmarking secure agentic code generation for fully homomorphic encryption. In *International Conference on Learning Representations*, volume 2026, pages 81257–81273, 2026.

[51] Mengxin Zheng, Lei Ju, and Lei Jiang. Cofhe: Software and hardware co-design for fhe-based machine learning as a service. *Frontiers in Electronics*, 3:1091369, 2023.

[52] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In *Proceedings of the 7th ACM workshop on encrypted computing & applied homomorphic cryptography*, pages 45–56, 2019.

[53] Hedong Zhang, Neusha Javidnia, Shweta Pardeshi, Qian Lou, and Farinaz Koushanfar. Cryptogen: Secure transformer generation with encrypted kv-cache reuse. *arXiv preprint arXiv:2602.08798*, 2026.

[54] Yukuan Zhang, Mengxin Zheng, and Qian Lou. Securerouter: Encrypted routing for efficient secure inference. *arXiv preprint arXiv:2604.15499*, 2026.

[55] Dacheng Li, Rulin Shao, Hongyi Wang, Han Guo, Eric P Xing, and Hao Zhang. Mpcformer: fast, performant and private transformer inference with mpc. *arXiv preprint arXiv:2211.01452*, 2022.

[56] Tianshi Xu, Lemeng Wu, Runsheng Wang, and Meng Li. Privcirnet: Efficient private inference via block circulant transformation. *Advances in Neural Information Processing Systems*, 37:111802–111831, 2024.

[57] Yuntian Chen, Zhanyong Tang, Tianpei Lu, Bingsheng Zhang, Zhiying Shi, and Zhiyuan Ning. Costsphere: A cost model-driven privacy-preserving machine learning framework with network context adaptation. *Cryptology ePrint Archive*, 2025.

[58] Neha Jawalkar, Nishanth Chandran, Divya Gupta, Rahul Sharma, and Arkaprava Basu. Matchmaker: Fast secure inference across deployment scenarios. *Cryptology ePrint Archive*, 2025.

[59] Yufan Zhu, Chao Jin, Khin Mi Mi Aung, and Xiaokui Xiao. Encformer: Secure and efficient transformer inference over encrypted data. *IEEE Transactions on Dependable and Secure Computing*, 2026.

[60] Jiaqi Xue, Mengxin Zheng, and Qian Lou. Robpi: Robust private inference against malicious client. *arXiv preprint arXiv:2602.19918*, 2026.

[61] Muhammad Husni Santriaji, Jiaqi Xue, Yancheng Zhang, Qian Lou, and Yan Solihin. Dataseal: Ensuring the verifiability of private computation on encrypted data. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 2378–2394. IEEE, 2025.

[62] Deevashwer Rathee, Anwesh Bhattacharya, Rahul Sharma, Divya Gupta, Nishanth Chandran, and Aseem Rastogi. Secfloat: Accurate floating-point meets secure 2-party computation. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 576–595. IEEE, 2022.

[63] Raghav Malik, Kabir Sheth, and Milind Kulkarni. Coyote: A compiler for vectorizing encrypted arithmetic circuits. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 118–133, 2023.

A Related works.

Hybrid HE-MPC systems [19, 27, 32, 33, 42] exploit the complementary costs of HE and MPC for private inference, while secure-computation compilers and cost-modeling frameworks such as EzPC [17], MP2ML [18] and nGraph-HE2 [52] show the importance of programmability and deployment-aware costs. Our work is complementary: rather than optimizing a single backend or using a fixed hybrid placement policy, we compile HE/MPC placement, bidirectional conversion, and HE maintenance decisions under a measured deployment profile.

CostSphere [57] is the closest prior system. It formulates protocol assignment as an ILP over a topologically linearized sequence of TOSA-level operators: conversion costs are charged only between adjacent positions, so cross-branch compatibility at residual and attention joins is not modeled. It approximates HE state by counting consecutive FHE operators and triggering a bootstrap after a threshold, whereas level consumption is heterogeneous (16 levels for Softmax vs. 1 for a matrix multiplication). Its costs come from offline regression models trained on one CPU platform with bandwidth features in the MPC model only, so the hardware dimension that reverses our optimal placement (Figure 5) is absent. CipherFlow instead plans exact per-level state on the branched graph with deployment-time measurements on CPU or GPU, in polynomial time.

Matchmaker [58] shares the view that no single configuration is optimal across deployments. It operates within 2PC MPC in the preprocessing model, choosing the fraction of a batch executed by a CPU-optimized LSS suite versus a GPU-based FSS suite to minimize batch completion time; it places no individual operators and has no HE domain. Its protocol suite could serve as an MPC backend profiled by MicroBench.

EncFormer [59], contemporaneous with this work, improves the hybrid stack itself (stage-compatible packing, a complex CKKS–MPC conversion, and communication-efficient nonlinear protocols) and selects FHE/MPC boundaries with offline cost equations over a small hand-enumerated design space. Its authors list cost-model-guided compilers that reduce hand-crafted placement as future work, which is the problem CipherFlow addresses. Its placement is the linear-in-HE, nonlinear-in-MPC family evaluated in Table 13.

Coyote [63] vectorizes encrypted arithmetic circuits within HE; batching-aware packing of this kind is complementary to cross-domain placement (Section 5).

B MicroBench Profiling Details

MICROBENCH is executed once for a target deployment profile before planning. It is not invoked per inference request. Given a model graph and a target backend, MICROBENCH extracts the distinct operator-shape patterns that appear in the graph and measures the secure primitive latency for each supported execution action. The resulting profile is cached and reused by the planner for all inputs under the same backend configuration.

MICROBENCH profiles five action types: (i) HE execution, (ii) MPC execution, (iii) HE→MPC conversion, (iv) MPC→HE conversion, and (v) HE maintenance through bootstrapping. (Detail in Table 7. The two conversion directions are measured separately because their compute, communication, and packing patterns are generally asymmetric.

Table 3: Refresh cost for a $[1, 128, 768]$ tensor (RTT $= 20$ ms): a direct bootstrap is compute-only, whereas the conversion round trip is communication-dominated, so the cheaper mechanism changes with the deployment.

Refresh mechanism	10 Mbps	100 Mbps	1 Gbps	3 Gbps
HE→MPC→HE round trip	63 s	6.33 s	0.672 s	0.252 s
Direct bootstrap, CPU (51.6 s)	**bootstrap**	round trip	round trip	round trip
Direct bootstrap, GPU (1.69 s)	**bootstrap**	**bootstrap**	round trip	round trip

Bootstrapping versus conversion-based refresh. A direct bootstrap involves no communication or interaction and is bound by compute hardware, whereas an HE→MPC→HE round trip is bound by

Table 4: Removing the direct-bootstrap transition (BERT-base). The planner may still refresh ciphertexts through HE→MPC→HE round trips.

Platform	BW	RTT	Full	−Bootstrap	Slowdown
GPU	10 Mbps	1 ms	472 s	1.83 ks	3.87×
GPU	10 Mbps	20 ms	472 s	1.83 ks	3.87×
GPU	10 Mbps	40 ms	472 s	1.83 ks	3.87×
GPU	10 Mbps	80 ms	472 s	1.83 ks	3.88×
GPU	100 Mbps	1 ms	472 s	582 s	1.23×
GPU	100 Mbps	20 ms	472 s	583 s	1.23×
GPU	100 Mbps	40 ms	472 s	584 s	1.24×
GPU	100 Mbps	80 ms	472 s	586 s	1.24×
CPU (all 16 points)	all		unchanged		1.00×
GPU, ≥1 Gbps	8 points		unchanged		1.00×

bandwidth and RTT. Table 3 shows that the cheaper mechanism flips within the evaluated grid on both platforms, and Table 4 shows the cost of removing direct bootstrapping from the planner.

Sensitivity to HE Level-Dependent Costs. Our HE profiles use NEXUS kernel costs measured at fixed levels. Table 5 measures these kernels across a 20-level chain on CPU and GPU (Section 4.1); for $N = 2^{13}$ and 2^{15}, these chains exceed the HE standard's 128-bit security bound and are used only for cost profiling. Each additional level increases non-linear kernel latency by 8–14%, while MatMul latency grows by $5.4\times$ on GPU and $20\times$ on CPU from level 1 to 20. Re-planning all 4×4 configurations with per-level costs and free modulus switching preserves HE/MPC placement across all 64 settings. CPU plans and predicted latencies remain unchanged, while GPU plans increase the median bootstrap count from 8.5 to 24 to execute linear layers at lower levels. Predicted GPU latency increases by at most 13% for BERT-base and 8% for ViT-B/16, with speedups over the best hybrid baseline of 2.16–14.69× and 2.34–15.94×, respectively (versus 2.42–16.53× and 2.52–17.21× under single-level costs). Without re-planning, the original plans are up to $2.5\times$ and $2.8\times$ slower under per-level costs. Accounting for the higher level consumption of NEXUS non-linear kernels (16/18/18 versus the assumed 8/3/4 for Softmax/LayerNorm/GeLU) leaves CPU plans unchanged and yields GPU speedups of 2.15–14.59× and 2.33–15.82× for BERT-base and ViT-B/16, respectively.

Table 5: Level-dependent HE cost of the NEXUS kernels behind the HE profiles. Each kernel keeps ring dimension and prime sizes on a 20-level chain; its input is encrypted at the top level and mod-switched to level ℓ before the timed region. δ: levels the planner charges / levels the kernel consumes. Columns give the runtime at input level ℓ relative to the level ℓ_{ref} at which the profile cost was measured (–: below the kernel's depth).

Kernel	Device	δ	ℓ_{ref}	1	2	4	8	12	16	18	20
MatMul	CPU	1 / 1	1	1.00	1.55	2.84	6.06	10.01	14.87		20.42
Softmax	CPU	8 / 16	18	–	–	–	–	–	0.74	1.00	1.29
LayerNorm	CPU	3 / 18	18	–	–	–	–	–	–	1.00	1.23
GeLU	CPU	4 / 18	18	–	–	–	–	–	–	1.00	1.27
MatMul	GPU	1 / 1	1	1.00	1.17	1.67	2.70	3.70	4.49		5.36
Softmax	GPU	8 / 16	16	–	–	–	–	–	1.00	1.20	1.39
LayerNorm	GPU	3 / 18	18	–	–	–	–	–	–	1.00	1.17
GeLU	GPU	4 / 18	18	–	–	–	–	–	–	1.00	1.15

C Limitation

CipherFlow currently optimizes single-request execution under fixed deployment profiles and does not account for concurrency, batching, queueing, or resource contention. Throughput optimization therefore requires a serving-level cost model. HE costs are profiled at a single level per operator; level-dependent costs can be incorporated into the planner but have not been measured. The current implementation also assumes static computation graphs and tensor shapes. Unseen shapes can be profiled and cached, but autoregressive decoding with evolving KV caches and data-dependent

routing in mixture-of-experts models require additional support. Finally, our security model assumes semi-honest two-party execution.

D Compiler Pseudocode

This appendix gives the algorithmic skeleton of the hardware-aware hybrid HE-MPC compiler. Algorithm 1 is the top-level driver: it consumes a model graph G and a deployment setting $\mathcal{D}$, calls MICROBENCH to instantiate a deployment-specific cost profile Π, decomposes G into SubBlock regions, computes a region transfer function T_R for each region by exact state-expanded planning, and finally composes regions through a macro-level shortest-path search over region boundary states. Algorithm 2 details the inner state-expanded shortest-path subroutine used in two places: *(i)* computing T_R for a SubBlock region, and *(ii)* composing across regions in the macro graph.

Algorithm 1 COMPILE: Hardware-Aware Hybrid HE–MPC Compilation

Require: Model graph $G = (V, E)$; deployment $\mathcal{D} = (\mathcal{H}, \mathrm{BW}, \mathrm{RTT})$; HE level budget L
Ensure: Optimized hybrid execution plan P^*

$\Pi \leftarrow \textsc{MicroBench}(G, \mathcal{H})$
$w \leftarrow \textsc{ComposeCost}(\Pi, \mathrm{BW}, \mathrm{RTT})$
$\mathcal{R} \leftarrow \textsc{ExtractSubBlock}(G)$

for all $R \in \mathcal{R}$ **do**
    $G_R^{\mathrm{state}} \leftarrow \textsc{BuildStateGraph}(R, L, w)$
    for all $(s_{\mathrm{in}}, s_{\mathrm{out}}) \in \mathcal{B}_R$ **do**
        $(T_R, \pi_R)[s_{\mathrm{in}}, s_{\mathrm{out}}] \leftarrow \textsc{StateEXP}(G_R^{\mathrm{state}}, s_{\mathrm{in}}, s_{\mathrm{out}})$
    end for
end for

$G^{\mathrm{macro}} \leftarrow \textsc{BuildMacroGraph}(G, \mathcal{R}, \{T_R\})$
$(C^*, \pi^*) \leftarrow \textsc{StateEXP}(G^{\mathrm{macro}}, s_{\mathrm{src}}, s_{\mathrm{dst}})$
$P^* \leftarrow \textsc{Materialize}(\pi^*, \{\pi_R\})$
return P^*

Algorithm 2 STATEEXP: State-Expanded Shortest Path

Require: State graph $\mathcal{G} = (\mathcal{S}, \mathcal{T}, w)$; source s_{src}; target s_{dst}
Ensure: Minimum cost C^* and predecessor map π

$C[s] \leftarrow \infty$ for all $s \in \mathcal{S}$
$C[s_{\mathrm{src}}] \leftarrow 0; \quad \pi \leftarrow \emptyset$
$Q \leftarrow \{(0, s_{\mathrm{src}})\}$
while $Q \neq \emptyset$ **do**
    $(c, s) \leftarrow \textsc{ExtractMin}(Q)$
    if $c > C[s]$ **then continue**
    if $s = s_{\mathrm{dst}}$ **then break**
    for all $(s, s') \in \mathrm{Out}(s)$ **do**
        if $\neg\textsc{Feasible}(s, s')$ **then continue**
        $c' \leftarrow c + w(s, s')$
        if $c' < C[s']$ **then**
            $C[s'] \leftarrow c'; \quad \pi[s'] \leftarrow s$
            $\textsc{Push}(Q, (c', s'))$
        end if
    end for
end while
return $C[s_{\mathrm{dst}}], \pi$

E StateExp

State. A state is $s = (i, d, \ell)$: operator index i along the linearized spine, domain $d \in \{\mathrm{HE}, \mathrm{MPC}\}$, and remaining level $\ell \in \{0, \ldots, L\}$ (canonically $\ell = L$ for MPC). Table 7 lists the five transitions. HE execution of v_i is feasible only if $\ell \geq \delta_i$ and v_i's HE kernel contract admits its input shape; otherwise the transition is absent, so infeasible placements are never generated. Conversions and bootstrapping do not advance i; MPC execution never changes ℓ.

Join handling. A SESE region consists of a fork u, a main branch, a skip branch, and a join v (a residual addition). For each entry state, BlockFold runs one state-expanded shortest-path search per branch, obtaining the cost of every reachable end state of each branch. For every pair of branch end states and each merge domain $d_v \in \{\mathrm{HE}, \mathrm{MPC}\}$ it (i) converts any branch whose domain differs from d_v; (ii) if $d_v = \mathrm{HE}$, aligns levels by one of four strategies: drop the higher-level operand to the lower level (free in CKKS), bootstrap the main branch, bootstrap the skip branch, or bootstrap both; (iii) executes v in d_v; and (iv) converts to the requested exit state. The minimum over all combinations is $T_R(s_{\mathrm{src}}, s_{\mathrm{dst}})$, and the argmin is stored for backtracking. Algorithm 3 summarizes the procedure.

Worked example (BERT encoder block). The block has 11 operators and two SESE regions: R_1 (fork QK, main branch Softmax→AV→Out-Proj, skip edge QK→Add_1) and R_2 (fork LN_1, main branch FFN_1 →GeLU→FFN_2, skip edge LN_1 →Add_2). With $L = 20$ there are $|\mathcal{B}| = 22$ boundary states (21 HE levels and MPC), so each region stores a 22×22 table. The macro spine is QK, R_1, LN_1, R_2, LN_2, and 12 blocks are chained by LN_2 →QK. On GPU at 10 Mbps every table entry that stays in HE is cheapest (bootstrapping 1.7 s vs. a 63 s round trip), and the planner outputs an all-HE plan with 17 bootstraps; on CPU at 3 Gbps the MPC-entry/MPC-exit entries dominate and the plan is all-MPC. Table 6 traces one block at an intermediate setting (GPU, 50 Mbps/20 ms, input arriving secret-shared) where a single block uses every action. QK stays in MPC, where its input already is; its output is converted to HE because MPC Softmax would transfer 106 MiB. At Add_1 the skip operand, still in MPC, is converted and dropped for free to the main branch's level 10. After LN_1 the rest of the block needs $1{+}4{+}1{+}3 = 9$ levels but only 7 remain, so the planner bootstraps (1.7 s) rather than taking an HE→MPC→HE round trip (12.6 s). At Add_2 both operands are aligned to the skip branch's level 7, and LN_2 leaves the block at level 4.

Table 6: Plan of one BERT-base encoder block (GPU profile, 50 Mbps / 20 ms, input arriving in MPC), one row per planner action with the state (d, ℓ) before and after it (levels without optional free drops). R_1, R_2: the two SESE regions; QK, LN_1, LN_2 lie on the macro spine.

Region	Operator	Action	Before	After	Cost (s)
	QK	MPC exec	(MPC)	(MPC)	3.28
R_1	input of Softmax	MPC→HE	(MPC)	(HE, 20)	12.61
R_1	Softmax	HE exec	(HE, 20)	(HE, 12)	0.26
R_1	AV	HE exec	(HE, 12)	(HE, 11)	3.75
R_1	Out-Proj	HE exec	(HE, 11)	(HE, 10)	3.75
R_1	skip operand of Add_1	MPC→HE	(MPC)	(HE, 20)	6.31
R_1	Add_1	HE exec	(HE, 10)	(HE, 10)	0.00
	LN_1	HE exec	(HE, 10)	(HE, 7)	0.72
R_2	input of FFN_1	bootstrap	(HE, 7)	(HE, 20)	1.69
R_2	FFN_1	HE exec	(HE, 20)	(HE, 19)	15.00
R_2	GeLU	HE exec	(HE, 19)	(HE, 15)	1.51
R_2	FFN_2	HE exec	(HE, 15)	(HE, 14)	3.75
R_2	Add_2	HE exec	(HE, 7)	(HE, 7)	0.00
	LN_2	HE exec	(HE, 7)	(HE, 4)	0.72
Total (planner)					53.4

F Complexity Analysis of BlockFold

We analyze the planning complexity before and after BlockFold. Let $G = (V, E)$ be the computation graph with $N = |V|$ operators. Let L denote the number of HE level states tracked by the planner. Since each operator can be assigned to either HE or MPC, and HE states additionally carry a level index, we write $q = O(L)$ for the number of cryptographic execution states per operator. Let J be

Algorithm 3 BLOCKFOLD: Region Transfer and Macro Composition

Require: Graph G; edge weights w; HE level budget L
Ensure: Region transfer tables $\{T_R\}$ and macro graph G^{macro}

$\mathcal{B} \leftarrow \{(\text{HE}, \ell)\}_{\ell=0}^{L} \cup \{(\text{MPC}, L)\}$
$\mathcal{R} \leftarrow \text{EXTRACTSESE}(G)$

for all $R = (u, m, k, v) \in \mathcal{R}$ **do**
  for all $s \in \mathcal{B}$ **do**
    $D_m \leftarrow \text{STATEEXP}(m, s)$
    $D_k \leftarrow \text{STATEEXP}(k, s)$
    for all $s' \in \mathcal{B}$ **do**
      $(T_R[s, s'], \pi_R[s, s']) \leftarrow \text{JOINOPT}(D_m, D_k, v, s')$
    end for
  end for
end for

$G^{\text{macro}} \leftarrow \text{COLLAPSE}(G, \mathcal{R}, \{T_R\})$
return $\{T_R, \pi_R\}, G^{\text{macro}}$

Table 7: State transitions in the expanded graph at operator v_i.

Transition type	State transition	Constraint	Cost
HE execution	$(i, \text{HE}, l) \to (i{+}1, \text{HE}, l - \delta_i)$	$l \geq \delta_i$	c_i^{HE}
MPC execution	$(i, \text{MPC}, L) \to (i{+}1, \text{MPC}, L)$	-	c_i^{MPC}
HE→MPC conversion	$(i, \text{HE}, l) \to (i, \text{MPC}, L)$	-	$c_i^{\text{HE}\to\text{MPC}}$
MPC→HE conversion	$(i, \text{MPC}, L) \to (i, \text{HE}, L)$	-	$c_i^{\text{MPC}\to\text{HE}}$
Bootstrapping	$(i, \text{HE}, l) \to (i, \text{HE}, L)$	-	c_i^{BS}

the number of join nodes in the graph, and let k_j denote the number of incoming branches of the j-th join.

Planning without BlockFold. Without BlockFold, the planner performs global state tracking over the original branched graph. At a join node, the planner must enforce compatibility among all incoming branch states: the domains must agree, and if the join is executed in HE, the incoming ciphertext levels must be aligned or made compatible through maintenance actions. If the j-th join has k_j incoming branches and each branch can end in one of q states, then the join induces $O(q^{k_j})$ joint state combinations.

A sequential chain with N operators would require $O(Nq)$ state nodes. However, each join introduces additional cross-branch state combinations. Since one branch state is already represented in the sequential state propagation, the j-th join contributes an additional multiplicative factor of q^{k_j-1}. Therefore, the worst-case global planning search space without BlockFold is

$$O\left(Nq \prod_{j=1}^{J} q^{k_j-1}\right).$$

When all joins have the same fan-in k, this simplifies to

$$O\Big(Nq^{1+J(k-1)}\Big).$$

Using $q = O(L)$, the complexity becomes

$$O\Big(NL^{1+J(k-1)}\Big).$$

Thus, even for binary joins ($k = 2$), naive global planning can grow as $O(NL^{1+J})$, which becomes impractical for residual and attention-heavy models with many joins.

Planning with BlockFold. BlockFold decomposes the graph into R single-entry single-exit (SubBlock) regions. Let n_r be the number of operators in the r-th region, so that $\sum_{r=1}^{R} n_r = N$. Instead of exposing all internal branch states to the global planner, BlockFold summarizes each region by a boundary-state transfer function. For each pair of entry and exit states, the planner computes the minimum internal cost of the region.

Each region has q possible entry states and q possible exit states, resulting in $O(q^2)$ boundary-state pairs. For a region of size n_r, the state-expanded shortest-path computation over all boundary pairs costs $O(n_r q^2)$. Therefore, the total cost of local region planning is

$$O\left(\sum_{r=1}^{R} n_r q^2\right).$$

After each region is summarized by its transfer function, the macro-level graph contains R region nodes, each with $O(q^2)$ transfer edges between boundary states. Macro-level composition therefore costs

$$O(Rq^2).$$

Combining local region planning and macro-level composition, the overall planning complexity with BlockFold is

$$O\left(\sum_{r=1}^{R} n_r q^2 + Rq^2\right).$$

Since $\sum_{r=1}^{R} n_r = N$, this becomes

$$O\big((N + R)q^2\big).$$

Substituting $q = O(L)$, we obtain

$$O\big((N + R)L^2\big).$$

Because $R \leq N$ for a valid decomposition into non-empty regions, the planning complexity is bounded by

$$O(NL^2).$$

Therefore, BlockFold reduces planning from exponential dependence on the number of joins in the worst case, $O(NL^{1+J(k-1)})$, to polynomial dependence on the number of HE states, $O(NL^2)$, while preserving exact optimization within each SubBlock region.

G Operator Placement under Hardware and Network Variation

Figure 5 reports the compiler-selected HE/MPC operator mix for BERT-base across bandwidth and RTT settings. The two platforms exhibit opposite placement patterns. On CPU, the compiler selects an MPC-dominant plan: at 10 Mbps it keeps only 12 of 132 operators in HE, and from 100 Mbps to 3 Gbps it selects a pure-MPC plan across all RTT values. This indicates that, under the CPU primitive profile, HE computation is sufficiently expensive that even communication-sensitive MPC becomes preferable once bandwidth is moderate.

GPU shows a qualitatively different pattern. At 10 and 100 Mbps, the compiler selects pure HE across all RTT values, reflecting the strong acceleration of HE operators and the high cost of MPC

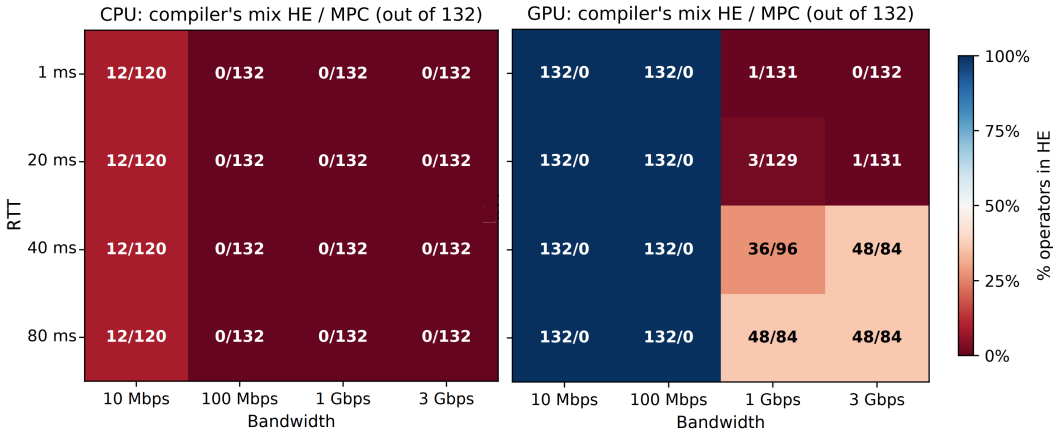


Figure 5: **Compiler-selected HE/MPC operator mix for BERT-base.** Each cell reports the number of operators placed in HE/MPC out of 132 operators, and color indicates the percentage of operators in HE. CPU and GPU exhibit opposite placement trends: CPU favors MPC once bandwidth is moderate, while GPU favors HE under bandwidth-constrained settings and introduces MPC only when communication becomes sufficiently cheap.

communication under constrained bandwidth. As bandwidth increases to 1-3 Gbps, the compiler begins to introduce MPC and mixed plans, with the exact mix depending jointly on bandwidth, RTT, and operator-level costs.

H Full Deployment Sweep

This appendix reproduces the complete 4×4 (bandwidth $\times$ RTT) sweep for both Transformer benchmarks side by side. BERT-base uses 12 encoder blocks at S=128 (132 operator instances); ViT-Base/16 uses 12 encoder blocks at S=197 (14×14 patches plus CLS token, 132 operator instances). All columns within a row share the same MicroBench primitive profile, so each comparison isolates placement; speedup is reported against the best static *hybrid* baseline (BumbleBee/BOLT/BLB-style placements) and the **Mix** column gives the compiler's HE/MPC operator split (out of 132). Latencies use "ks" = 10^3 s for compactness.

The BERT-base tables additionally report the published BumbleBee system (BB^{sys}) plotted in Figure 4(a,b) and the placement-fair speedup; the submitted version mistakenly repeated the ViT CPU table in place of the BERT-base CPU table. Table 13 summarizes the per-operator baselines over the full grid.

I MicroBench Measurement Methodology

This appendix describes how MICROBENCH constructs the deployment-specific cost profile used by the compiler. The purpose of MICROBENCH is to instantiate operator-level costs for the target hardware and network setting, rather than to benchmark a complete hybrid system for every possible deployment.

Cost entries. For each operator, conversion, and HE maintenance action, MICROBENCH records three quantities: local compute time c^{loc}, communication volume b, and round count r. The local compute time is measured on the target CPU or GPU backend. The communication volume and round count are obtained from the corresponding HE/MPC protocol implementation and message-size accounting. Given a deployment setting $\mathcal{D} = (H, \text{BW}, \text{RTT})$, the compiler evaluates each action as

$$c_{\mathcal{D}} = c_H^{\text{loc}} + \frac{8b}{\text{BW}} + r \cdot \text{RTT}.$$

Table 8: BERT-base, CPU, full 4×4 sweep. BB^{sys} is the published BumbleBee end-to-end system measurement recomposed under each network (the orange bars in Figure 4); BB^{map} applies BumbleBee's placement rule to our primitives. Speedup is against the fastest baseline including BB^{sys} (as plotted); $\text{Speedup}^{\text{fair}}$ uses placement-fair mappings only.

BW	RTT	BB^{sys}	BB^{map}	BOLT	BLB	Ours	Speedup	$\text{Speedup}^{\text{fair}}$	Mix
10 Mbps	1 ms	10.8 ks	16 ks	14.4 ks	12.8 ks	7.93 ks	1.36×	1.62×	12/120
10 Mbps	20 ms	22 ks	16 ks	14.4 ks	12.9 ks	7.95 ks	1.62×	1.62×	12/120
10 Mbps	40 ms	33.7 ks	16.1 ks	14.5 ks	12.9 ks	7.97 ks	1.61×	1.61×	12/120
10 Mbps	80 ms	57.2 ks	16.1 ks	14.5 ks	12.9 ks	8 ks	1.61×	1.61×	12/120
100 Mbps	1 ms	2.67 ks	9.18 ks	9.02 ks	8.86 ks	1.27 ks	2.11×	6.99×	0/132
100 Mbps	20 ms	13.8 ks	9.19 ks	9.02 ks	8.86 ks	1.3 ks	6.84×	6.84×	0/132
100 Mbps	40 ms	25.6 ks	9.19 ks	9.03 ks	8.87 ks	1.33 ks	6.69×	6.69×	0/132
100 Mbps	80 ms	49 ks	9.2 ks	9.04 ks	8.88 ks	1.39 ks	6.41×	6.41×	0/132
1 Gbps	1 ms	1.86 ks	8.49 ks	8.48 ks	8.46 ks	457 s	4.06×	18.50×	0/132
1 Gbps	20 ms	13 ks	8.5 ks	8.48 ks	8.47 ks	486 s	17.43×	17.43×	0/132
1 Gbps	40 ms	24.7 ks	8.51 ks	8.49 ks	8.47 ks	516 s	16.42×	16.42×	0/132
1 Gbps	80 ms	48.2 ks	8.52 ks	8.5 ks	8.48 ks	576 s	14.73×	14.73×	0/132
3 Gbps	1 ms	1.79 ks	8.44 ks	8.44 ks	8.43 ks	397 s	4.52×	21.22×	0/132
3 Gbps	20 ms	12.9 ks	8.45 ks	8.44 ks	8.44 ks	426 s	19.81×	19.81×	0/132
3 Gbps	40 ms	24.7 ks	8.45 ks	8.45 ks	8.44 ks	456 s	18.52×	18.52×	0/132
3 Gbps	80 ms	48.2 ks	8.47 ks	8.46 ks	8.45 ks	516 s	16.38×	16.38×	0/132

Table 9: BERT-base, GPU, full 4×4 sweep. BB^{sys} is the published BumbleBee end-to-end system measurement recomposed under each network (the orange bars in Figure 4); BB^{map} applies BumbleBee's placement rule to our primitives. Speedup is against the fastest baseline including BB^{sys} (as plotted); $\text{Speedup}^{\text{fair}}$ uses placement-fair mappings only.

BW	RTT	BB^{sys}	BB^{map}	BOLT	BLB	Ours	Speedup	$\text{Speedup}^{\text{fair}}$	Mix
10 Mbps	1 ms	10.8 ks	10.9 ks	9.34 ks	7.75 ks	472 s	16.41×	16.41×	132/0
10 Mbps	20 ms	22 ks	11 ks	9.36 ks	7.76 ks	472 s	16.44×	16.44×	132/0
10 Mbps	40 ms	33.7 ks	11 ks	9.37 ks	7.78 ks	472 s	16.47×	16.47×	132/0
10 Mbps	80 ms	57.2 ks	11 ks	9.41 ks	7.81 ks	472 s	16.53×	16.53×	132/0
100 Mbps	1 ms	2.67 ks	1.46 ks	1.3 ks	1.14 ks	472 s	2.42×	2.42×	132/0
100 Mbps	20 ms	13.8 ks	1.48 ks	1.32 ks	1.16 ks	472 s	2.45×	2.45×	132/0
100 Mbps	40 ms	25.6 ks	1.49 ks	1.33 ks	1.17 ks	472 s	2.48×	2.48×	132/0
100 Mbps	80 ms	49 ks	1.53 ks	1.36 ks	1.2 ks	472 s	2.55×	2.55×	132/0
1 Gbps	1 ms	1.86 ks	515 s	498 s	482 s	97.8 s	4.93×	4.93×	1/131
1 Gbps	20 ms	13 ks	530 s	513 s	497 s	126 s	3.94×	3.94×	3/129
1 Gbps	40 ms	24.7 ks	546 s	529 s	512 s	148 s	3.45×	3.45×	36/96
1 Gbps	80 ms	48.2 ks	578 s	560 s	542 s	186 s	2.92×	2.92×	48/84
3 Gbps	1 ms	1.79 ks	444 s	439 s	433 s	38 s	11.39×	11.39×	0/132
3 Gbps	20 ms	12.9 ks	460 s	454 s	448 s	66.7 s	6.71×	6.71×	1/131
3 Gbps	40 ms	24.7 ks	476 s	469 s	463 s	90.5 s	5.11×	5.11×	48/84
3 Gbps	80 ms	48.2 ks	508 s	500 s	493 s	127 s	3.87×	3.87×	48/84

This decomposition separates hardware-dependent local computation from network-dependent communication cost, allowing the planner to re-evaluate the same operator profile under different bandwidth and RTT settings.

Profiling protocol. We discard warm-up iterations and report the mean over timed iterations. Table 12 summarizes the profiling protocol used by MICROBENCH. HE operators and conversions are measured with wall-clock timing, while GPU MPC operators use CUDA-event timing to reduce timing overhead for short GPU kernels.

Shape-keyed reuse. MicroBench profiles operators by shape rather than by layer identity. Each entry is keyed by

$$(\text{op_type}, \text{input_shape}, \text{output_shape}, \text{backend}).$$

Table 10: ViT-B/16, CPU, full 4×4 sweep. Bold marks the best RTT per bandwidth (the value shown in Figure 4).

BW	RTT	BumbleBee	BOLT	BLB	Ours	Speedup	Mix HE/MPC
10 Mbps	1 ms	28.2 ks	25.4 ks	22.6 ks	13 ks	**1.75**×	12/120
10 Mbps	20 ms	28.2 ks	25.4 ks	22.6 ks	13 ks	1.74×	12/120
10 Mbps	40 ms	28.2 ks	25.4 ks	22.6 ks	13 ks	1.74×	12/120
10 Mbps	80 ms	28.2 ks	25.4 ks	22.7 ks	13 ks	1.74×	12/120
100 Mbps	1 ms	15.9 ks	15.6 ks	15.3 ks	2.05 ks	**7.48**×	0/132
100 Mbps	20 ms	15.9 ks	15.6 ks	15.3 ks	2.08 ks	7.38×	0/132
100 Mbps	40 ms	15.9 ks	15.6 ks	15.3 ks	2.11 ks	7.27×	0/132
100 Mbps	80 ms	15.9 ks	15.6 ks	15.4 ks	2.17 ks	7.08×	0/132
1 Gbps	1 ms	14.7 ks	14.6 ks	14.6 ks	739 s	**19.75**×	0/132
1 Gbps	20 ms	14.7 ks	14.6 ks	14.6 ks	768 s	19.02×	0/132
1 Gbps	40 ms	14.7 ks	14.6 ks	14.6 ks	798 s	18.32×	0/132
1 Gbps	80 ms	14.7 ks	14.7 ks	14.6 ks	858 s	17.05×	0/132
3 Gbps	1 ms	14.6 ks	14.6 ks	14.5 ks	642 s	**22.65**×	0/132
3 Gbps	20 ms	14.6 ks	14.6 ks	14.6 ks	671 s	21.70×	0/132
3 Gbps	40 ms	14.6 ks	14.6 ks	14.6 ks	701 s	20.78×	0/132
3 Gbps	80 ms	14.6 ks	14.6 ks	14.6 ks	761 s	19.15×	0/132

Table 11: ViT-B/16, GPU, full 4×4 sweep. Bold marks the best RTT per bandwidth (the value shown in Figure 4).

BW	RTT	BumbleBee	BOLT	BLB	Ours	Speedup	Mix HE/MPC
10 Mbps	1 ms	19.1 ks	16.3 ks	13.5 ks	789 s	17.13×	132/0
10 Mbps	20 ms	19.1 ks	16.3 ks	13.5 ks	789 s	17.15×	132/0
10 Mbps	40 ms	19.1 ks	16.3 ks	13.5 ks	789 s	17.17×	132/0
10 Mbps	80 ms	19.1 ks	16.3 ks	13.6 ks	789 s	**17.21**×	132/0
100 Mbps	1 ms	2.54 ks	2.26 ks	1.99 ks	789 s	2.52×	132/0
100 Mbps	20 ms	2.55 ks	2.28 ks	2 ks	789 s	2.54×	132/0
100 Mbps	40 ms	2.57 ks	2.29 ks	2.01 ks	789 s	2.55×	132/0
100 Mbps	80 ms	2.6 ks	2.32 ks	2.04 ks	789 s	**2.59**×	132/0
1 Gbps	1 ms	888 s	860 s	832 s	161 s	**5.17**×	1/131
1 Gbps	20 ms	904 s	875 s	847 s	189 s	4.48×	3/129
1 Gbps	40 ms	920 s	891 s	862 s	216 s	3.99×	36/96
1 Gbps	80 ms	952 s	922 s	892 s	259 s	3.44×	36/96
3 Gbps	1 ms	766 s	757 s	747 s	62 s	**12.05**×	0/132
3 Gbps	20 ms	781 s	771 s	761 s	90.7 s	8.39×	0/132
3 Gbps	40 ms	797 s	787 s	776 s	121 s	6.43×	3/129
3 Gbps	80 ms	829 s	818 s	806 s	164 s	4.92×	48/84

Thus, repeated Transformer blocks reuse the same profile entry whenever they share the same operator type and tensor shape. This reduces profiling overhead and ensures placement-fair comparison: all static baselines and our compiler consume the same cached cost entry for the same operator shape under the same deployment profile.

Representative cost entries. Table 14 reports representative entries used by the planner. These entries illustrate the cost asymmetry that motivates hardware-aware placement. HE operators often have high local compute cost but no MPC-style communication rounds, while MPC operators are locally cheaper but introduce communication volume and round-trip latency. Cross-domain conversions also contribute non-negligible cost, so the compiler must jointly optimize operator placement and conversion placement.

Table 12: **MicroBench profiling protocol.** We discard warm-up iterations and report the mean over timed iterations.

Primitive family	Warmups	Repeats	Timing method
HE operators	1	3	wall-clock
HE↔MPC conversions	2	5	wall-clock
MPC operators on CPU	1	3	wall-clock
MPC operators on GPU	1	3	CUDA event

Table 13: Per-operator baselines over the 4×4 grid (min–max latency ratio to CIPHERFLOW; all plans scored by the same evaluator). EncFormer-style placement keeps linear operators in FHE and runs Softmax/LayerNorm/GELU in MPC with complex-packed (half-payload) conversions; under our profiles it coincides with the BLB column.

Model	Platform	Greedy	Random best	Random median	EncFormer-style
BERT-base	CPU	1.02–1.37×	1.63–10.92×	1.86–16.31×	1.61–21.22×
BERT-base	GPU	1.01–5.75×	1.74–14.02×	2.08–18.58×	2.42–16.53×
ViT-B/16	CPU	1.02–1.43×	1.64–11.34×	1.92–17.02×	1.74–22.65×
ViT-B/16	GPU	1.02–5.39×	1.88–13.94×	2.27–18.57×	2.52–17.21×

J Cost-Model Validation

We executed the five nonlinear and attention operators of one encoder block (B=1, S=16) through the real SCI/BOLT bridges, both in isolation and back to back, and compared measured protocol wall-clock with the composed prediction $c^{\mathrm{loc}} + 8b/\mathrm{BW} + r \cdot \mathrm{RTT}$. Networks were emulated with Linux `tc netem` on the loopback device. Figure 6 shows the result. The attention bridges run their 12 per-head products on 12 connections but count traffic on two, so their byte counts are scaled by 6. On localhost the chain is within 8–10% of the prediction, with Softmax, LayerNorm, and GeLU within $\pm 8\%$ and the two attention matrix multiplications 43–44% faster than predicted. Under shaped networks the chain-level prediction is conservative (measured 52–58% and 41–44% below prediction at 1 Gbps/4 ms and 100 Mbps/20 ms). The error comes from the round term: the protocol round counter reports about 1,000 rounds for Softmax, but the implied blocking round trips are only about 175, because most rounds are pipelined; charging raw rounds overestimates round-heavy operators. We therefore refine the network term: for each operator, an effective volume b_{eff} and effective blocking rounds r_{blk} are fitted (non-negative least squares) to its 1 Gbps and 100 Mbps runs, and the operator is charged $c^{\mathrm{loc}} + 8b_{\mathrm{eff}}/\mathrm{BW} + r_{\mathrm{blk}} \cdot \mathrm{RTT}$. The fit gives 175 and 132 blocking rounds for Softmax and LayerNorm (counted: 1,024 and 856), none for the transfer-bound QK, AV, and GeLU, and volumes within 10% of the counted ones except Softmax (14.5 vs. 24.6 MB). On runs not used in the fit, the chain error falls from 52–58% to 1–11% at 1 Gbps and from 41–44% to 1–7% at 100 Mbps; at 10 Mbps/40 ms the per-operator error is within 3% for QK, AV, and LayerNorm, while Softmax is under-predicted (18.9 vs. 27.5 s, one repetition).

K Implementation Details

HE backend. HE operators call NEXUS CKKS kernels through Python adapters. Kernel shape and packing contracts (e.g., LayerNorm on $[B, S, 768]$ without affine parameters, packed $[3, B, S, 768]$ attention inputs with 12 heads) are checked explicitly; a violated contract marks the HE transition infeasible rather than silently approximating. **MPC backend.** MPC operators run in external SCI/BOLT or SHAFT processes for both parties; for SCI/BOLT, communication bytes and rounds are read from the protocol implementation's own counters, so (b, r) are deterministic and only local compute is timed; for SHAFT, local compute is timed per operator and the bytes and rounds measured over a full SHAFT run are apportioned to operators by compute time. The SCI linear bridge caps its batch dimension, and larger shapes are tiled. **Conversions.** HE→MPC uses mask-and-decrypt and MPC→HE uses re-encryption; the directions are asymmetric and are profiled separately per tensor shape. **Profiling.** Warm-up iterations are discarded, short GPU kernels are timed with CUDA events, and profiles are cached by (operator, input shape, output shape, backend) so repeated blocks are profiled once.

Table 14: **Representative MicroBench cost entries** (BERT-base, S=128; conversions for a $[1, 128, 768]$ tensor). Each entry reports local compute time c^{loc}, communication volume b, and round count r; these are the exact values consumed by the planner. †Derived from the CPU entries with published NEXUS-CUDA speedups (MatMul CPU/20×, bootstrapping CPU/30×), because profiling them on the GPU ran out of memory.

Operator / action	Domain / backend	c^{loc}	b	r
Attention_QK	HE / CPU	1.55e5 ms	0	0
Attention_QK	MPC / CPU	211 ms	6.2 MiB	1
Softmax	HE / CPU	5.98e4 ms	0	0
Softmax	MPC / CPU	2.72e3 ms	79.8 MiB	11
GeLU	MPC / CPU	1.05e3 ms	30.9 MiB	4
HE→MPC	Conv. / CPU	10.7 ms	37.5 MiB	1
MPC→HE	Conv. / CPU	10.5 ms	37.5 MiB	1
Bootstrap	HE / CPU	5.16e4 ms	0	0
Attention_QK†	HE / GPU	7.5e3 ms	0	0
Attention_QK	MPC / GPU	11.7 ms	19.1 MiB	3
Softmax	HE / GPU	264 ms	0	0
Softmax	MPC / GPU	64.7 ms	106.2 MiB	15
GeLU	MPC / GPU	13.5 ms	22.2 MiB	3
HE→MPC	Conv. / GPU	1.38 ms	37.5 MiB	1
MPC→HE	Conv. / GPU	1.27 ms	37.5 MiB	1
Bootstrap†	HE / GPU	1.68e3 ms	0	0

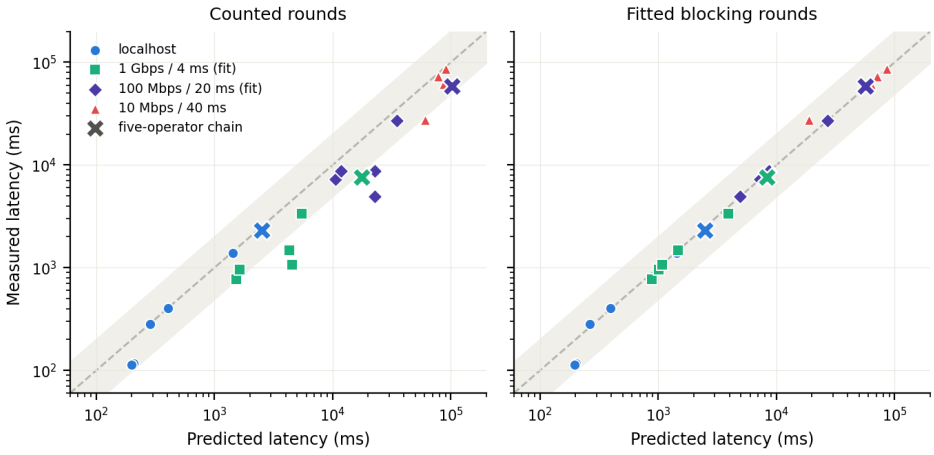


Figure 6: Predicted versus measured latency of real SCI/BOLT protocol execution, per operator and for the five-operator chain (crosses), under localhost and `tc netem`-shaped networks, with counted rounds (left) and fitted blocking rounds (right). The fit uses the 1 Gbps and 100 Mbps per-operator runs; shading marks a factor of 2.

L Communication Volume

Table 15: Online HE–MPC protocol communication (GiB per BERT-base inference) of the latency-optimal plan versus static hybrid placements.

Platform / network	Ours latency	Ours	BumbleBee	BOLT	BLB	All-MPC
GPU, 10 Mbps / 1 ms	472 s	0.0	12.3	10.4	8.5	10.5
GPU, 3 Gbps / 80 ms	127 s	9.4	12.3	10.4	8.5	10.5
CPU, 10 Mbps / 1 ms	7.93 ks	7.9	8.9	7.0	5.2	10.5
CPU, 1 Gbps / 20 ms	486 s	10.5	8.9	7.0	5.2	10.5

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes], [No], or [N/A].
- [N/A] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for [N/A]).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will also be asked to include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While [Yes] is generally preferable to [No], it is perfectly acceptable to answer [No] provided a proper justification is given (e.g., error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering [No] or [N/A] is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS Paper Checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and Section 1 state the four contributions (hardware-aware formulation, MicroBench, state-expanded planning, BlockFold) and the up-to $16.5\times$ / $22\times$ speedups; these are supported by Section 2.3, Section 3, and the experiments in Section 4 and Appendix H.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

Answer: [Yes]

Justification: The Discussion at the end of Section 4 explicitly notes that CIPHERFLOW optimizes single-request execution under a fixed deployment profile and does not yet model multi-client concurrency, batching, queueing, or shared-resource contention.

Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

   Justification: The threat model and problem formulation are stated in Section 2.1–2.3; the BlockFold complexity result is derived in full in Appendix F, and the state-transition rules used by the planner are listed in Appendix E.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: Section 4.1 specifies models, sequence lengths, hardware, network sweep, and HE/MPC backends; the algorithmic skeleton is given in Appendix D, and the MicroBench profiling protocol and representative cost entries are reported in Appendix I.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: The code link is in `https://github.com/loopleap-lab/CipherFlow`.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.

- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 4.1 reports HE level budget, polynomial degree, sequence lengths, BW/RTT sweep, and the static baselines; we evaluate inference latency only (no training), so optimizer/data-split fields do not apply.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Explicit error bars are omitted because per-operator HE/MPC latency is dominated by deterministic protocol-level compute and communication, and the run-to-run variance is much smaller than the order-of-magnitude speedups we report.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [Yes]

Justification: Section 4.1 specifies the compute platforms (AMD Ryzen Threadripper PRO 3955WX 16-core CPU; NVIDIA B200 GPU); end-to-end inference latency is reported in Section 4.2 / Appendix H, and compilation time is reported in Table 2.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

Answer: [Yes]

Justification: The work does not involve human subjects, personal data, or scraped content; it uses public model architectures and standard cryptographic primitives, and we have reviewed and conformed to the NeurIPS Code of Ethics.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: Section 1 motivates the work by privacy regulation and recent data-leakage incidents; by lowering the latency cost of HE/MPC inference, CIPHERFLOW makes privacy-preserving deployment more practical (positive impact). We do not foresee a direct negative pathway, since the compiler operates on user-supplied models and produces only protocol-level execution plans.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.

- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: We do not release any pretrained generative models, image generators, or scraped datasets; the released artifact is a compiler that produces secure-inference execution plans, which carries no high misuse risk.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We cite the original works for all reused assets, including the HE/MPC backends (NEXUS-CKKS, SCI/SHAFT) in Section 4.1 and the static hybrid baselines (BumbleBee, BOLT, BLB) in Section 1 and Section 4. All assets are used within the terms of their original licenses.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes]

Justification: The new asset (the CIPHERFLOW compiler) is documented in the paper itself: Section 3 describes the compilation pipeline, Appendix D provides the algorithmic skeleton, Appendix E lists the state-transition rules, and Appendix I documents the MicroBench profiling methodology and representative cost entries.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A]

Justification: The work does not involve crowdsourcing or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: The work does not involve human subjects, so IRB approval is not required.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [N/A]

Justification: BERT and ViT are evaluated only as fixed inference workloads for the compiler; LLMs are not part of the core methodology, and any incidental use during writing did not affect scientific content.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.