

SING: Improving the Efficiency of MPC Protocol Assignment using Graph Neural Networks

(Full Version)*

Jannis Blüml* Moritz Huppert* Nora Khayata† Joachim Schmidt† Thomas Schneider†

Technical University of Darmstadt, Darmstadt, Germany

* {firstname}.{lastname}@tu-darmstadt.de

† {khayata,joachim.schmidt,schneider}@encrypto.cs.tu-darmstadt.de

Abstract

Secure Multi-Party Computation (MPC) enables private computation, but has significantly higher overhead than plaintext execution. Hybrid MPC compilers improve concrete efficiency by mapping distinct computation parts to contextually optimal MPC protocols. However, state-of-the-art systems like Silph (Chen et al., S&P'23) depend on deployment-specific cost models that are cumbersome to retune, and compute mappings via brittle heuristics or costly Integer Linear Programming (ILP), limiting scalability and portability across protocols and deployment settings.

We present SING, the first machine-learning-based framework for hybrid MPC share assignment. SING leverages Graph Neural Networks (GNNs) for: (1) imitation of Silph's assignments, accelerating share assignment by up to 76,697× with comparable quality; and (2) cost-driven learning, where we train a GNN cost predictor on synthetic or empirical costs (e.g., runtime or communication), freeze it, and train the share-assigning GNN to minimize predicted costs. The latter supports expressive non-linear cost models, avoiding ILP's linearity constraints, and enables retargeting to new protocol suites and deployment settings by re-fitting the predictor. Finally, we release our synthetic benchmark resources, including a dataset of 704 MPC circuits with wide-ranging hybrid assignments.

1 Introduction

The rapid digitalization of our economies and societies introduces unprecedented security and privacy risks. As interconnectivity grows, so does the demand for joint computation over sensitive data subject to user privacy concerns, business competition, or regulatory constraints such as the GDPR [92]. Prominent examples include detecting money laundering across financial institutions [94], and monitoring the state of a pandemic using data from multiple healthcare providers [68]. General-purpose, privacy-preserving technologies with strong cryptographic foundations for such applications have existed for decades [40, 76, 100], but incur a substantial performance overhead in realistic settings.

A versatile primitive for joint privacy-preserving computation is Secure Multi-Party Computation (MPC) [40, 100]. In an MPC protocol, multiple parties $P_1, \ldots, P_N$ jointly compute a function f on their private inputs $x_1, \ldots, x_N$ such that no party learns anything beyond what is implied by the output. This generic interface enables the redesign of many digital processes as systems that are private by design and provably secure. Applications range from private Machine Learning (ML) inference [39, 70, 77] and training [53, 54, 56] to secure auctions [14, 15], private set intersection [50], satellite collision avoidance [47], and private password monitoring [5].

Over the past decades, an ecosystem of MPC protocols has emerged [40, 64, 78, 85, 100, 102]. However, no single protocol is uniformly optimal across workloads, network conditions, and performance metrics [31, 86]. As a result, relying on any single protocol leads to unnecessarily poor performance for general-purpose computation. This observation motivated hybrid/mixed-protocol designs, wherein different parts of a computation are executed using different MPC protocols [13, 17, 19, 31, 48, 88, 97]. Since finding such partitions by hand is tedious and error-prone, share assignment frameworks aim to automatically partition the computation and map each partition to an appropriate protocol [22, 26, 52, 57, 80]. Since the ILP-based framework Silph [26] has been shown to outperform prior work [22, 52], we use it as the primary baseline throughout the paper. These existing share assignment frameworks face two fundamental limitations in terms of *extensibility* and *scalability*.

First, they depend on protocol-specific cost models that estimate the runtime and/or communication of executing

*Please cite the conference version published at USENIX Security'26 [12]. During the thorough artifact review for functionality and reproducability, the reviewers identified an issue with Experiment E2 (§8.5) that required us to revise the reported results. Specifically, our original implementation did not ensure that the MPC benchmarking circuits used to compare SING (this work) and Silph [26] were excluded from SING's training pipeline. We therefore revised the experiment by retraining the SING models while explicitly excluding the corresponding benchmarking circuits from the respective datasets. In this revision of the paper, we report the updated experimental results.

each operation under each MPC protocol, including conversions. These models are typically fitted from extensive microbenchmarks and then treated as fixed. While, in principle, they can be re-fit for new hardware, network conditions, or protocol implementations, doing so in practice is labor-intensive and requires substantial protocol expertise (e.g., enumerating supported primitives and conversions, and selecting appropriate circuit or program features).

Second, given a cost model, current systems compute share assignments via local heuristics [34, 57, 80], or via global optimization formulations, e.g., ILP/MILP [26, 57], LP with totally unimodular constraints [52], or dynamic programming [22]. Heuristic approaches typically scale well, but provide no optimality guarantees and can become brittle when new primitives or conversions are introduced. In contrast, mathematical-programming-based approaches often do not scale well and yield assignments that are optimal only with respect to a *simplified* objective encoded in the formulation (e.g., additive, linear costs) [22, 26, 52, 57]. These simplifications are frequently necessary for tractability; however, they can mispredict the true runtime and communication costs of concrete MPC implementations, thereby reducing the reliability of the resulting assignments. In particular, simplified objectives often fail to capture cross-operation effects that dominate real executions, such as batching, parallelism, and scheduling. This gap is consistent with the fact that accurate cost modeling is itself a challenging systems problem. CostCO [34], for instance, learns an empirical regression model from real executions, but still requires protocol expertise to define a meaningful feature set $\mathcal{F}$ and relies on carefully designed benchmark circuits to fit its parameters. Even then, the model is restricted to a quadratic form, illustrating the effort to reliably predict observed MPC behavior [34].

Our Contributions

SING: ML-based share assignment. We introduce *SING (Share assIgnments Neurally Generated)*, the first machine-learning-based framework focused on *scalable* hybrid share assignment. SING supports three training regimes: (i) supervised learning of Silph's [26] assignments to obtain ILP-quality assignments at inference-time speed, (ii) cost-guided training that minimizes a learned surrogate of Silph's objective, and (iii) cost-guided training that minimizes a learned surrogate of *measured* execution costs of ABY [31].

Learned cost models as training surrogates. To enable cost-guided learning, we train neural cost predictors for two targets: C^{Silph}, which distills Silph's tractable cost objective, and C^{Runtime}, which is trained on ABY [31] measurements to approximate runtime and communication. We then treat the learned predictor as a differentiable objective and apply gradient-based optimization to obtain high-quality hybrid assignments. Cost-guided optimization modularizes training into two independent steps and thus enables fully end-to-end, black-box training of share assignment, provided a high-quality cost predictor C^{Runtime} is available. Consequently, unlike the supervised setting, training the share-assignment model does not require solving large (M)ILP instances. Conceptually, our approach can be extended to new settings by refitting the models to a new environment or protocol suite using freshly collected MPC runtime measurements. This may be simpler than tuning analytical cost models with new microbenchmarks for every operation–protocol combination, particularly as prior work [26] does not always specify how cost parameters were chosen. However, for a new protocol suite, only the circuits in the dataset remain directly reusable; the existing share assignments no longer apply. Training the cost surrogate would therefore require first generating a broad distribution of new assignments, which remains challenging.

Dataset and reproducible pipeline. We release the first dataset of 704 MPC circuits designed for learning-based optimization, which we use for both cost prediction and share assignment. It covers realistic workloads and structural diversity, with multiple candidate share assignments per circuit. The dataset combines 54 established MPC benchmarks, 200 randomly generated circuits, and 450 tailored Large Language Model (LLM)-generated programs. All circuits are compiled to a common representation, cost-labeled under multiple protocol mappings, and split into train/validation/test with balanced assignment distributions. Together with our dataset, we permanently release our codebase on Zenodo.[1]

Empirical evaluation on hybrid compilation. On Silph [26] benchmark circuits, SING accelerates share assignment by 17.1 ×–76,696.5 × while producing assignments with competitive runtime and communication. By reducing both the engineering effort for cost modeling and the computational overhead of share assignment, SING brings the compilation of hybrid MPC protocols, and thus secure computation, closer to practical deployment.

Previous work has conjectured that optimal hybrid MPC compilation is NP-hard [52]. Thus, practical compilers face a trade-off between efficient heuristics and globally optimal methods with worst-case runtime exponential in the circuit size, which may not terminate within reasonable time limits. We therefore adopt a heuristic, learning-based approach and, like prior heuristics, provide no optimality guarantees. However, our method learns its heuristics from data with less reliance on expert knowledge, potentially capturing complex rules that are difficult to encode manually while retaining efficient inference. Although any such heuristic cannot outperform the global optimum, it may closely approximate optimal

[1] https://doi.org/10.5281/zenodo.20347719

baselines at substantially lower and therefore practical runtime cost. Our design also inherits the limited interpretability of graph neural networks.

2 Preliminaries

2.1 Graph Neural Networks

Graph Neural Networks (GNNs) extend deep learning to non-Euclidean domains. Given a graph $G = (V, E)$ with node features $\phi(v) \in \mathbb{R}^{d_v}$, GNN layers iteratively update each node's representation by aggregating information from its neighbors. *Convolutional* GNNs generalize the notion of convolution to irregular structures: node embeddings are updated via learnable transformations applied to aggregated neighbor features. Let $\mathcal{N}(i) = \{ j \in V \mid (i,j) \in E \}$ denote the (one-hop) neighborhood of node i. A canonical example is the Graph Convolutional Network (GCN) [59]: $x'_i = \sigma\left(\mathbf{W} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{x_j}{\sqrt{\delta_i \delta_j}}\right)$, where δ_j is the degree of node j, including self-loops, and σ a non-linearity. For directed graphs, Dir-GNN [84] applies the update on both original and reversed edges, and combines the results. For graph-level predictions, node embeddings from the final layer are aggregated via global pooling. See [20, 44] for broader overviews of GNNs and geometric deep learning.

We use `GraphSAGE` [45], which scales to large and sparse graphs via neighborhood sampling and inductive aggregation: for each node, it samples a fixed-size subset $\mathcal{S}(i) \subseteq \mathcal{N}(i)$ and applies a learned aggregator to compute embeddings, enabling generalization to unseen nodes or graphs. Using the mean aggregator, the weight update function is: $x'_i = \sigma\left(\mathbf{W_1} x_i + \mathbf{W_2} \cdot \text{mean}_{j \in \mathcal{S}(i)} x_j + \mathbf{b}\right)$. GraphSAGE is well-suited for MPC circuit graphs due to its efficiency and ability to generalize to unseen graph structures.

2.2 Multi-Party Computation Protocols

Secure Multi-Party Computation (MPC) protocols allow multiple parties $P_1, \ldots, P_N$ to compute a known function f on their private inputs $x_1, \ldots, x_N$. The two most well-known protocols are Yao's Garbled Circuits (GCs) [100] and GMW [40]. We briefly describe the concepts behind both protocols and examine the trade-offs involved in selecting one over the other. We refer to ABY [31] for a full overview of the optimizations in modern implementations of both protocols. Following previous work [22, 26, 34], we consider $N = 2$ parties, but note that our approach is general. A prerequisite for both protocols is Oblivious Transfer (OT) [81]. OT enables a *receiver* P_2 to obliviously select a bitstring m_b from two bitstrings m_0, m_1 provided by the *sender* P_1, such that b is not revealed to P_1 and m_{1-b} is not revealed to P_2.

2.2.1 Yao's Garbled Circuits

Yao's Garbled Circuits (GCs) [100] is a secure two-party protocol for evaluating Boolean circuits while hiding all intermediate wire values in the circuit. In the following sections, we denote this protocol by `yao`. The wire values 0 and 1 are represented as wire labels v_0, v_1, which are bitstrings of length κ, the symmetric security parameter. The protocol involves the roles of a Garbler P_G and an Evaluator P_E.

First, P_G samples the wire labels, constructs the garbled operations, and sends these to P_E. A *garbled* operation in the circuit processes wire labels and outputs a new wire label corresponding to the plaintext value of the operation result. Then, using OT, P_E obliviously learns the wire labels according to its inputs. With the input wire labels and the garbled operations, P_E evaluates the entire circuit locally. Using state-of-the-art optimizations [64, 85, 102], party P_E evaluates `XOR` gates with simple bit-wise operations, while `AND` gate evaluation requires symmetric cryptographic operations.

As GCs only operate on Boolean values, all operations on integer values need to be evaluated using their potentially more costly Boolean circuit counterparts (e.g., a multiplication circuit has size quadratic in the operand bit-length [63]). The number of communication rounds is constant and unaffected by the number of gates or circuit depth. Instead, the total number of `AND` gates is the main performance indicator; thus, minimizing the number of `AND` gates is the main optimization objective for GCs [36, 91].

2.2.2 GMW

Goldreich-Micali-Widgerson (GMW) [40] is a general-purpose MPC protocol, which computes functions over integer rings $\mathbb{Z}_{2^k}$ or the Boolean finite field $\mathbb{F}_2$. We denote these two modes as different protocols: `arith` for **arith**metic ($\mathbb{Z}_{2^k}$) and `bool` for **Bool**ean ($\mathbb{F}_2$) computation. Both modes use the same underlying techniques to compute additions or multiplications in the respective domains, and conversions are necessary when moving between them. An intermediate value in the circuit x is secret-shared among parties $P_1, \ldots, P_N$ as $[x] = (x_1, \ldots, x_N)$. Each party P_i owns a share x_i such that $x = \sum_{i=1}^{N} x_i$. Reconstructing a value in plaintext from its shared form requires one round of communication, in which all parties send their shares to every other party.

GMW supports the two fundamental operations over the underlying ring or field: addition and multiplication. To add secret-shared values x and y, parties simply add their local shares, requiring no communication. Multiplication is more involved and uses a precomputed one-time Beaver triple (a, b, c) such that $a \cdot b = c$ [6], with a, b, and c secret-shared before protocol execution. Evaluating a product also requires reconstructing a masked intermediate value, which incurs one round of communication; the masking ensures that no information about the underlying intermediates is revealed.

As a result, GMW requires a number of communication rounds that is linear in the function's multiplicative depth, which can slow execution in high-latency networks [31, 86]. In return, GMW typically uses less memory and communication volume than Yao's garbled circuits, since a Boolean wire is represented by a single bit rather than a κ-bit label. Consequently, performance optimization in GMW primarily focuses on reducing the number of multiplications and, in particular, the multiplicative depth.

2.2.3 Circuit Representation

MPC frameworks [9, 17, 31, 32, 48, 55, 73] take as input a description of the function to be securely evaluated, which we model as a *circuit* c. Although representations differ across frameworks (e.g., due to supported operations or constructs like loops), most can be expressed in the classical form: $c = (V, E, o) \in \Gamma$, where $V = \{1, \ldots, n\}$ is the set of *nodes*, $E \subseteq V \times V$ the set of *edges*, and $o : V \to \Lambda$ assigns an operation type to each node. The graph $G = (V, E)$ is a Directed Acyclic Graph (DAG), and Γ denotes the set of all such circuits.

This representation is directly compatible with GNNs (cf. §2.1): the circuit graph $G = (V, E)$ serves as the input graph and node features are given by $\phi := o$. We consider the following operation set Λ, also supported by Silph [26]:

- `ADD`, `SUB`, `MUL`, `DIV`, `REM`, `EQ`, `GT`, `LT`, `GE`, `LE`: signed 32-bit integer arithmetic and comparisons.
- `AND`, `OR`, `XOR`, `NOT`: logical operations on 32-bit integers or Booleans. In `yao` and `bool`, 32-bit bitwise operations are applied to all bits in parallel.
- `SHL`, `LSHR`: left shift and logical right shift on 32-bit integers; the shift amount is known at compile time.
- `SELECT`, `STORE`: array read and write. `SELECT` returns the element at a given index; `STORE` returns a copy with one element replaced.
- `IN`, `OUT`: circuit inputs (Boolean or 32-bit integer) and outputs (exempt from share assignment).

2.2.4 Protocol Mixing

GMW [40] (`arith` and `bool`) and garbled circuits [100] offer complementary advantages. In low-latency settings, arithmetic sharing is typically most efficient for arithmetic-heavy, low-depth subcircuits (notably multiplications), Boolean sharing excels for multiplexer-heavy components, especially under parallelism, and comparisons are often best handled in `yao` [31]. In high-latency environments, by contrast, the additional interaction rounds of `arith` and `bool` can dominate, making `yao`'s constant-round online phase preferable, especially for deep (Boolean) computations [31], such as Universal Circuits for Private Function Evaluation [60, 65, 69, 93]. These trade-offs motivate *hybrid compilation* (or *protocol mixing*), where subcircuits are assigned to the protocol/sharing that best matches their operation mix and deployment environment [13, 19, 22, 26, 48, 52, 57, 80, 88].

Let $c = (V, E, o)$ be a circuit (cf. §2.2.3). Mixed-protocol execution is specified by a *share assignment* $s : V \to P$ that maps each node $v \in V$ to a protocol $p \in P$ in which v is evaluated. Following previous work [22, 26, 31, 52, 80], we consider $P = \{\texttt{arith}, \texttt{bool}, \texttt{yao}\}$. Not every mapping s is feasible, since protocols support different operations (e.g., `arith` cannot directly evaluate bitwise `XOR`); we therefore write $\Psi \subseteq (V \to P)$ for the set of valid share assignments.

Whenever an edge $(v_1, v_2) \in E$ connects nodes assigned to different protocols, i.e., $s(v_1) \neq s(v_2)$, the output of v_1 must be converted into the representation expected by v_2. These conversions incur protocol-dependent costs and can be highly asymmetric: for example, ABY shows that converting from `yao` to `bool` can be done locally, whereas converting from `bool` to `yao` requires OT [31]. Consequently, good hybrid assignments must balance per-protocol gate costs against the overhead of conversions. Especially in low-latency environments, conversion overheads can be small enough that switching sharings can pay off for very small subcircuits (e.g., a single multiplication) [31]. Hence, effective hybrid compilation should be tailored to the targeted deployment environment and concrete circuit structure as optimal choices vary greatly.

2.3 LLMs for Dataset Generation

Large Language Models (LLMs) learn to predict and generate text from large corpora and are often based on the Transformer architecture [95]. Interaction typically begins with a user-provided *prompt*, to which the model generates a *response*; additional prompts can be supplied iteratively, enabling conversational use. LLMs have been used for large-scale data collection in ML projects, e.g., the model `StarCoder2-Instruct` [98] was trained on instruction-code pairs with instructions generated using `StarCoder2-15B`. Similarly, `InverseCoder` [99] uses LLMs to generate instruction-code pairs. In the context of circuit learning, Chen et al. [27] propose LLMs as a potential source of training data for circuit models, particularly when specialized to hardware description languages rather than imperative languages; RTLCoder [72] is one such example.

Besides random circuit generation and established benchmarking suites, we use LLMs to add a wide range of *meaningfully structured* MPC circuits into our circuit dataset instead of purely random instances (cf. §5.3). However, despite their usefulness, LLMs can be unreliable and produce *hallucinated* output that is invalid, duplicate, or otherwise unsuitable for inclusion in a dataset [71]. To mitigate this, prior work like The Stack [62] employs dataset *decontamination*. Because MPC circuits impose additional structural and semantic constraints, we develop custom prompting and decontamination procedures instead of directly reusing existing pipelines.

3 Related Work

We first provide an overview of MPC frameworks (§3.1). A key step in mixed-protocol share assignment is to identify the lowest-cost share assignment w.r.t. a given cost model. Accordingly, we review prior work on MPC cost modeling (§3.2) and mixed-protocol share assignment (§3.3).

3.1 MPC Frameworks

There are numerous MPC frameworks, dating back to the early 2000s, that initially focused on implementing collections of protocols designed to run *in isolation*. The very first MPC framework was Fairplay [73], which implements Yao's GC protocol for two parties, including a compiler and the protocol runtime. Since then, many other implementations of GCs followed, extending support to more parties [10], supporting various security models [90, 96], and compressing GCs using hardware synthesis tools [30, 51, 91]. Besides highly efficient MPC protocol implementations, MPC compilers have been developed in tandem with MPC frameworks with the goal of improving the accessibility and experience of developing functions for MPC [1, 18, 23, 36, 37, 67, 73, 75, 79, 91].

Moving toward a *hybrid setting*, multiple MPC protocols are combined within a single computation to improve runtime and reduce communication overhead, as first done by TASTY [48], which mixed GCs and additively homomorphic encryption. ABY [31] combines arithmetic and Boolean GMW [40] with GCs [100], including pairwise conversions between all protocols for secure two-party computation. ABY has since been extended to function-dependent preprocessing [21, 78], and utilized in subsequent MPC research works [22, 24, 26, 36, 42, 79] and applications [8, 13]. ABY is commonly used as the MPC backend for (automatic) share-assignment frameworks, such as OPA [52] and Silph [26]. We also utilize ABY for this purpose.

MOTION [17] is a mixed-protocol MPC framework implementing the ABY protocols for any number of parties via multi-party GMW [40], BMR [7], and conversions. Garg et al. [38] designed a system for efficient hybrid MPC with many parties. Their framework includes efficient protocols based on Boolean and arithmetic secret sharing, GCs, and Homomorphic Encryption (HE). With this, Garg et al. present the first framework that extends the ABY protocols with HE, achieving high efficiency. Due to the modularity of SING, cost prediction and automatic share assignment can be performed independently of any specific MPC framework. Consequently, supporting other frameworks, e.g., MOTION [17] or Garg et al. [38] could be achieved, which we leave as future work.

MP-SPDZ [55] includes a multitude of protocols based on GCs and secret sharing for any number of parties. In contrast to ABY [31], MP-SPDZ does not support conversions between GCs and secret sharing; conversions between arithmetic and Boolean sharing are done via (e)daBits [33]. SCALE-MAMBA [2] is similar to MP-SPDZ, but focuses on maliciously secure MPC protocols. SEEC [32] implements a memory-efficient runtime for (mixed) secret sharing-based protocols. It proposes a compact in-memory representation to reduce the memory footprint of secret-sharing-based protocols. EzPC [24] is a mixed-protocol MPC framework focused on privacy-preserving machine learning applications. In contrast to other works, the original paper proposes a simple heuristic for hybrid compilation that always executes multiplications in the arithmetic domain. Internally, EzPC uses ABY [31], EMPToolkit [96], and other function secret-sharing based protocols [43, 54].

When only two protocols are mixed, as in MP-SPDZ [55], SCALE-MAMBA [2], EzPC [24], and SEEC [32], an optimal mixing can already be computed in polynomial time using Linear Programming (LP) [52]. Consequently, more sophisticated approaches such as Silph [26] and SING are unnecessary in this setting and instead become relevant when mixing ≥ 3 protocols, as in ABY [31], MOTION [17], or Garg et al.'s system [38]. For three or more protocols, finding an optimal assignment becomes NP-hard [52].

3.2 Modeling Costs of MPC

Cost models estimate runtime/communication for a given circuit and protocol(s) without performing the computation in MPC. They are necessary to evaluate highly localized decisions, like protocol selection for a specific subcircuit during hybrid compilation, since executing the circuit in MPC for every candidate decision would be prohibitively expensive.

Schröpfer and Kerschbaum [87] were the first to propose an analytical *Performance Model*, considering Garbled Circuits (GCs) [100] and additively Homomorphic Encryption (HE) [76]. Circuits are evaluated only in one protocol, so protocol mixing was not considered.

CostCO [34] casts MPC cost modeling into a quadratic regression problem. Users provide a set of *features* $\mathcal{F}$ to CostCO, which represent factors that significantly influence protocol performance. Specifying such $\mathcal{F}$ meaningfully requires expert knowledge of the underlying MPC protocol. Additionally, features that supposedly do not impact performance are disregarded (e.g., the number of XOR gates), and only linear combinations of features are considered due to the nature of quadratic regression. Finding the coefficients for the cost model requires samples of protocol executions. For this, CostCO benchmarks randomly generated circuits, where generation follows specific rules. While CostCO includes a proof-of-concept hybrid compiler that utilizes its generated cost models to synthesize hybrid MPC protocols, its protocol selection relies on a simple randomized greedy heuristic, primarily to demonstrate the utility of the cost models.

XENON [61] is also a (linear) regression-based cost model, which adds more consideration to network behavior, e.g., the effects of TCP congestion control on network latency. How-

ever, XENON does not consider protocol mixing.

CostSphere [28] explores hybrid protocols that combine MPC with fully homomorphic encryption (FHE) for privacy-preserving machine learning. When FHE is among the available protocols, the cost is non-linear and globally coupled to protocol assignment: assigning some parts to FHE may require larger FHE parameters, thereby changing the cost of subsequent FHE segments and conversions. To predict these costs, CostSphere trains several regression models (including Linear Regression, Random Forest, and XGBoost) and reports that XGBoost performs best overall.

Existing cost models are often difficult to apply to hybrid compilation: models for running an entire circuit under a single protocol do not extend to mixed protocol execution, and searching over all possible subcircuit–protocol assignments is prohibitively expensive or relies on heuristics that embed deployment assumptions. To overcome this, SING learns execution costs directly from empirical measurements across diverse hybrid assignments (third training regime) and uses gradient-based optimization to automatically find high-quality share assignments. As a result, SING is protocol-agnostic and data-driven, adapts to different networks and platforms, and integrates cleanly into existing MPC toolchains.

3.3 Share Assignment based on Cost Models

Most prior work on hybrid share assignment [22,26,52,57,80] begins with a simple additive cost model: it sums per-operation costs of the selected protocol and adds conversion costs when necessary. The cost parameters are typically obtained via micro-benchmarks. The resulting objective is then minimized using techniques such as ILP, greedy search, or dynamic programming. The advantage of this simple additive model over more complex models (cf. §3.2) is that it keeps the optimization problem tractable.

Kerschbaum et al. [57] study share assignment for MPC functions when the available protocol choices are additively Homomorphic Encryption (HE) [76] and garbled circuits (GCs) [100]. They propose two assignment methods: an exact Integer Linear Programming (ILP)-based formulation and a heuristic strategy. The heuristic follows a hill-climbing procedure: it starts with an all-GC assignment and then scans the circuit, switching individual operations to HE whenever the cost model predicts an improvement. This process repeats until no further local change reduces the cost. As a greedy local search, the heuristic is not guaranteed to find a globally optimal assignment; however, the authors report that the greedy heuristic's runtime is generally close to the ILP optimum.

Similarly, CheapSMC [80] proposes heuristic share assignment over the three ABY [31] protocols. For each circuit, it runs three greedy heuristics and returns the assignment with the lowest predicted cost: a *bottom-up* pass that processes nodes in topological order and chooses the cheapest protocol (including conversion costs) given already-fixed inputs, a *top-down* variant that applies the same rule in reverse order, and a *hill-climbing* heuristic initialized with all-GC.

HyCC [22] considers the same three ABY protocols and extends the CBMC-GC [36] compiler with automated hybrid selection. It first partitions the computation into coarse-grained (multi-gate) modules that are expected to share the same protocol and derives a parallel execution schedule for these modules. Given this schedule and benchmark-based per-operation and conversion costs, HyCC then applies dynamic programming (DP) over the module graph to choose, for each module, whether to use the `arith`, `bool` (GMW), or `yao` implementation and where to insert conversions, minimizing the predicted objective globally with respect to the cost model.

OPA [52] proposes an ILP formulation, similar to [57]. Unlike prior formulations, OPA's ILP is designed to be relaxed to Linear Programming (LP), enabling efficient (polynomial-time) optimization. This efficiency comes with a restriction: the formulation applies only to assignments over two protocols. To handle larger protocol sets, OPA solves the problem for every pair of protocols and selects the cheapest result, so the final assignment uses at most 2 protocols.

Silph [26] also computes share assignment via ILP, but unlike OPA, it directly supports assignments over all three ABY protocols, rather than restricting them to protocol pairs and relying on an LP relaxation. To remain scalable despite the lack of such a relaxation, Silph avoids solving a single monolithic ILP: it combines ILP solving with graph/hypergraph partitioning, uses a hierarchical optimization over partitions, and exploits call-site similarity to reuse solutions across similar contexts. Built on top of the CirC circuit compiler [75], Silph adds an MPC backend that solves the ILP and emits the resulting mixed-protocol circuit. A separate executor then loads this output and runs it using ABY [31].

A major limitation of ILP- and DP-based approaches is that, while they can compute an optimal assignment, the optimum is only with respect to the underlying (and typically simplified) cost approximation. More broadly, there is an inherent tension between accurate cost modeling and the tractability of optimization. Our first two training regimes, i.e., emulating Silph's optimal assignments and distilling Silph's additive cost model, inherit this limitation, even as they demonstrate the feasibility of ML-based cost modeling and share assignment. In contrast, our third training regime mitigates it by training directly on empirical MPC execution costs, rather than relying on a synthetic, simplified cost model.

4 Technical Overview: How to SING

SING consists of three main pieces, which we illustrate in Fig. 1. The Dataset Collection pipeline (§5) generates labeled MPC circuits with share assignments and cost labels, 2) the Cost Prediction Model (§6) estimates the execution cost of a circuit-assignment pair, and the Share Assignment Model (§7) infers protocol assignments for new circuits.

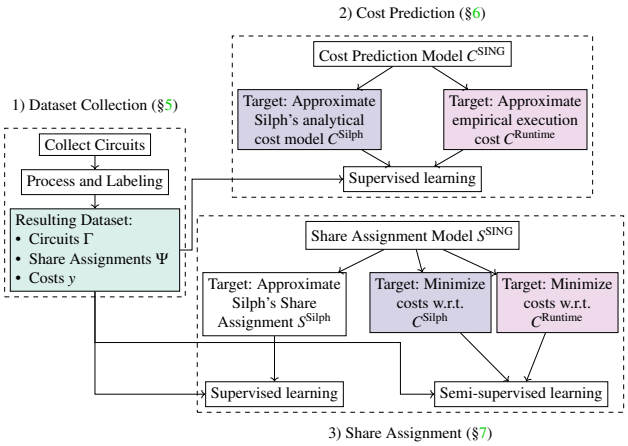


Figure 1: Overview of our SING framework and approach for ML-based MPC protocol assignment, consisting of MPC Dataset Collection (§5), MPC Cost Prediction (§6), and MPC Share Assignment (§7).

1) Dataset Collection (§5) Our circuit dataset comprises 704 circuits, which were collected using various methods. We began by incorporating existing benchmark MPC circuits (cf. §5.1) from previous work [26, 46] and common cryptographic tasks [3, 74]. These circuits alone are insufficient for achieving meaningful machine learning outcomes. Hence, we have expanded our dataset by generating random circuits (cf. §5.2) with additional restrictions during generation for enhanced quality. In addition, we have utilized large language models (LLMs) to generate MPC-compatible C code (cf. §5.3), which we have subsequently processed to make them suitable for MPC. After collecting the circuits, we have processed and labeled them (cf. §5.4) in accordance with the goals of cost prediction and share assignment. We then prepare our training/validation/test dataset splits.

2) Cost Prediction (§6) Finding an optimal share assignment aims to minimize the cost of MPC execution, for instance, runtime, memory consumption, or online communication. This requires a model that can accurately predict the cost of a given circuit–share-assignment pair (c,s). Our longer-term goal is to adopt a holistic approach that transcends per-protocol cost models, as in [22, 26, 34], towards an ML-based solution that generalizes to MPC beyond ABY [31].

To this end, we train graph neural networks for cost estimation: (a) $y = C^{\text{Silph}}(c,s)$, which approximates Silph's analytical cost model [26], and (b) $y = C^{\text{Runtime}}(c,s)$, which predicts empirically measured execution costs when running circuits in ABY [31]. We start with C^{Silph} because its simplicity keeps Silph's ILP formulation tractable, enabling us to validate our training pipeline for both cost prediction and share assignment by directly comparing ML-predicted costs and assignments against Silph's results. Since our approach does not rely on ILP-based optimization, we are not limited to simplified additive models and can instead move to a cost predictor that more faithfully captures real-world costs in the target deployment, C^{Runtime}. Moreover, this measurement-driven training methodology applies to any MPC framework for which execution traces can be collected.

3) Share Assignment (§7) To train an effective share-assignment model, we design and evaluate three training regimes, yielding three models: (a) fully supervised training that directly imitates Silph's assignments [26]; (b) semi-supervised training that learns assignments by minimizing $y = C^{\text{Silph}}(c,s)$; and (c) semi-supervised training that learns assignments by minimizing $y = C^{\text{Runtime}}(c,s)$. The two semi-supervised variants differ only in the cost predictor used to define the learning objective.

As with cost prediction, we begin by imitating Silph to validate the training pipeline and establish a strong baseline, directly comparing ML-predicted share assignments against the optimal assignments produced by Silph. Even this imitation model improves scalability, since inference no longer requires solving an ILP instance. However, it is inherently limited to reproducing Silph's decisions rather than independently minimizing cost. Ultimately, we must move beyond imitation to train an ML-based share-assignment model that is fully independent of prior solvers. We therefore transition to semi-supervised learning, where the model independently learns to identify assignments that yield lower costs.

5 SING Circuit Dataset

To the best of our knowledge, there is no prior MPC circuits dataset designed for learning-based optimization. We therefore construct our own. Our dataset is used to supervise both cost prediction (§6) and share assignment (§7) and includes realistic MPC workloads, structural diversity, and various share assignments per circuit. Our dataset consists of 54 MPC benchmark circuits (§5.1), 200 randomly generated circuits (§5.2), and 450 LLM-generated circuits (§5.3). All circuits are compiled to a common representation, assigned multiple candidate protocol mappings, labeled with cost, and split into train/validation/test with balanced assignment distributions.

5.1 MPC benchmark circuits

We include circuits from existing MPC toolchains to capture realistic workloads. The open-source Silph repository [26] provides benchmark circuits used in its evaluation. These include tasks such as classification, convolutional layers, and basic arithmetic subroutines. We ingest all benchmarks and all available size variants (e.g., different CNN input configurations). We also include smaller test circuits from Silph.

We further add circuits from LLVM for MPC [46], which compiles high-level code via LLVM [66] to MPC-friendly

circuits. The public release contains four benchmarks, which we adapt with minimal changes to match Silph's input syntax. Finally, we introduce a few hand-selected circuits to improve coverage of bitwise and arithmetic workloads: ChaCha (from OpenSSL [74]), array permutation, common bit-twiddling primitives [3], and a linear classifier $y = \mathsf{Sign}(wx+b)$. These emphasize control-heavy logic, multiplexing, and Boolean structure that cannot be represented purely in `arith`.

5.2 Random circuits

Past work on MPC cost models uses random circuits to fit regressors (e.g., CostCO [34]). We likewise include randomized instances to increase structural diversity.

Rather than sampling arbitrary DAGs, we generate random C programs from a restricted, typed grammar and then compile them to circuits. This preserves compilation invariants of our MPC representation (e.g., `MUX` nodes with correctly wired select bits and branches), avoids degenerate graphs that naive DAG sampling can produce, and lets us control coarse statistics (e.g., operator mix and circuit depth) so the synthetic distribution remains close to the benchmark circuits. This helps reduce overfitting to specific layouts and improves generalization to unseen but valid structures, consistent with noise-as-regularizer theory [11]; surveys report similar benefits broadly [89], particularly on graphs [103].

The generator expands productions until a budget b is exhausted, counting each arithmetic/Boolean operation as one unit. We generate 100 circuits using $b = 10^4$ and further 100 using $b = 10^5$. The circuit sizes range from 28 to 1456 operations ($\mu = 607.64,\ \sigma = 365.41$), all compiled with Silph.

5.3 LLM-generated circuits

To extend the variety of our dataset, we generate additional circuits using large language models (LLMs). LLMs are well-suited for quickly producing many variants of small, standalone computational kernels, which lets us explore a broader distribution of circuit motifs than MPC benchmarks and random circuits alone. We prompt both hosted models (ChatGPT [25] and DeepSeek [29]), and locally run open-source LLMs via `ollama`, using task descriptions that constrain the output to MPC-compatible C code. Constraints include: no floating-point arithmetic and `goto`s, static loop bounds, and no unrestricted pointer aliasing. We also employ a few-shot style prompting strategy, where we present the model with samples of valid input/output pairs and request variants.

From an initial pool of 8557 generated programs, only 450 survive the filtering process. We discard samples that fail to compile or are duplicates. The size of the accepted LLM-generated circuits ranges from 2 to 81,442 operations ($\mu = 375.14, \sigma = 4120.48$).

5.4 Processing pipeline and labeling

The raw dataset contains C source files that are converted into graph-structured circuits and labeled across several stages.

5.4.1 Compilation to circuits.

We compile each C source file using Silph [26]. Silph inlines all functions and runs its ILP-based optimizer with the `empirical` cost model and `smart_glp` selection scheme, producing (i) a mixed-protocol circuit and (ii) a corresponding share assignment. The output is stored in text form; we hash the input C file and output circuit file to deduplicate circuits.

5.4.2 Alternative share assignments.

To effectively train a mixed-protocol cost predictor, we require multiple candidate share assignments for the *same* compiled circuit. Hence, we generate additional assignments by iterating over all operations and sampling a valid protocol for each node uniformly at random, subject to the typing constraints of §2.2.4. We generate up to $\max(3, \lceil \phi \log_2 n \rceil)$ assignments per circuit, where $n = |V|$ is the number of operations and $\phi = 3$ by default. We reject duplicate and invalid assignments.

5.4.3 Perturbation of share assignments.

To broaden coverage over realistic circuits and avoid learning a model that merely memorizes benchmark layouts, we employ *perturbation*. Starting from an existing globally optimal share assignment (e.g., the assignment produced by Silph's ILP formulation), we generate additional samples via small, controlled edits while preserving validity. We consider two perturbation types: (a) *single-node* changes that switch the protocol of one randomly selected node (e.g., `arith` $\rightarrow$ `bool` for a multiplication), and (b) *localized multi-node* changes that switch the protocol of several nearby nodes within the same neighborhood. We implement (b) using the neighbor-sampling technique introduced in GraphSAGE [45].

From an ML perspective, share assignment perturbation constitutes *graph-level data augmentation*: it increases sample density around realistic instances rather than introducing unrelated random graphs. Such augmentation is known to improve robustness and reduce overfitting to specific graph topologies [83, 101]. Thus, perturbed assignments encourage the model to learn cost-relevant structural patterns rather than memorizing exact circuit layouts.

5.4.4 Cost labels

Each (circuit, assignment) pair (c, s) is labeled with a scalar cost y. We support two sources of ground truth:

1. The analytical cost function from Silph [26], $C^{\text{Silph}}(c, s)$, ported into our pipeline.

2. Empirical measurements (runtime or communication), obtained by executing the circuit c with assignment s in ABY [31]. We emulate different network settings using an adaptation of NEON [61].

This enables training against either Silph's cost model or directly against observed execution costs in our second and third training regimes, respectively. A detailed discussion of these two choices is provided in §6.

5.4.5 Train/validation/test splitting

Training, validation, and test splits must be balanced not only in size, but also in their distribution of protocol assignments. We therefore implement a custom greedy splitting heuristic. We iterate over circuits in descending size and assign each circuit to the split that optimizes $\texttt{Score}_t(c) = \alpha\texttt{SizeScore}_t(c) + (1-\alpha)\texttt{DistScore}_t(c)$, where $\texttt{SizeScore}_t$ penalizes deviation from the desired fraction of total samples for split t, and $\texttt{DistScore}_t$ penalizes deviation from the overall protocol distribution. We use $\alpha = 0.3$ in all experiments.

6 SING Cost Prediction Model

The cost prediction model estimates the circuit's execution cost for a given share assignment. Let P be the set of available MPC protocols and Λ the set of supported operations. A circuit is represented as $c = (V, E, o)$, where V is the set of operations, $E \subseteq V \times V$ the dataflow edges, and $o : V \to \Lambda$ maps each node to its operation type. A share assignment is a mapping $s \in \Psi \subseteq (V \to P)$ that assigns each node to a protocol. The cost model is a function $C : \Gamma \times \Psi \to \mathbb{R}$, which predicts a scalar cost for the pair (c, s). We refer to the measured runtime as C^{Runtime}, to Silph's analytical estimate as C^{Silph} [26], and to our learned model as C^{SING}_θ with parameters θ. We treat cost prediction as a *graph-level regression* task (§2.1). Each node $v \in V$ is represented by a feature vector $\phi_C(v) = \texttt{OneHot}(o(v)) \| \texttt{OneHot}(s(v))$, where $d_v = |\Lambda| + |P|$. This encodes both the operation performed and the protocol under which it will run. Intuitively, this allows the model to learn the impact of protocol boundaries, conversions, and per-operation cost.

Architecture and Training. Our cost model is a lightweight two-layer GraphSAGE network [45] with 16 hidden channels, followed by a global additive pooling layer. Each node's features are first embedded and aggregated over its neighborhood, then mapped to a scalar contribution. The circuit-level cost estimate is the sum of these node-level contributions. The model has $|\theta| = 849$ trainable parameters.

We evaluated directional message passing (Dir-GNN [84]) and found it unnecessary for this task: whether a node incurs a conversion cost is determined by the protocols assigned to its inputs, which are already visible along in-going edges.

We train C^{SING} in a supervised manner on pairs (c, s) from our dataset. Each training example consists of the graph (V, E), the node features ϕ_C derived from o and s, and a scalar label y. The label can be instantiated in two ways:

1. $y = C^{\text{Silph}}(c, s)$: learn to approximate Silph's [26] analytical cost model. Empirically, we obtained the best results when restricting the share assignments s to assignments generated by Silph [26] (without perturbations) and uniformly random assignments.

2. $y = C^{\text{Runtime}}(c, s)$: learn to approximate empirical execution costs (e.g., runtime or communication when executing in ABY [31]). For estimating WAN runtimes, our dataset comprises Silph-generated assignments and perturbations thereof, excluding random assignments.

We minimize the mean squared error (MSE) with standard gradient-based training: $\mathcal{L}(\theta, x, y) = \texttt{MSE}(\theta, x, y)$.

7 SING Share Assignment Models

The share assignment model predicts, for a new circuit, which protocol each node should be evaluated under. Formally, let $\Psi \subseteq (V \to P)$ be the set of valid assignments under the protocol-mixing rules of §2.2.4. We define $S : \Gamma \to \Psi$, where $S(c)$ returns an assignment s mapping each node of c to a protocol in P. We use S^{Silph} to denote the ILP-based assignment used by Silph [26], and S^{SING}_θ for our learned model.

We frame the share assignment as a *node-level classification* task. Each node $v \in V$ is represented by $\phi_S(v) = \texttt{OneHot}(o(v))$, where $d_v = |\Lambda|$. The model outputs, for each node, a distribution over P.

7.1 Architecture

Our assignment model is a two-layer Directed Graph Neural Network (Dir-GNN) [84]; message-passing layers are implemented as GraphSAGE [45] blocks. We use 64 hidden channels and a final softmax over the protocol set, for a total of $|\theta| = 8{,}201$ parameters.

Intuitively, predicting the best protocol for a node depends on both its inputs *and* its consumers (i.e., the nodes following this node). For example, if a node's output feeds subsequent nodes assigned to `yao`, choosing `arith` locally may introduce an expensive conversion. Directional aggregation in Dir-GNN allows each node to access information propagated along both incoming and outgoing edges, capturing these constraints.

Because the model predicts each node's protocol independently, the raw output might not satisfy the validity constraints of §2.2.4. During deployment, we apply a post-processing step: any node assigned an invalid protocol is reassigned to `yao`, which is always valid in our setting. We do *not* correct

assignments during training, so the model is still encouraged to learn valid assignments directly.

7.2 Training Approaches

We present three training setups with progressively less reliance on prior MPC share-assignment frameworks: starting with imitation of Silph's ILP-based assignments and ending with a model trained to minimize measured MPC runtimes.

7.2.1 SING 1: Supervised training with Silph [26]

In the supervised setting, we train S^{SING} against share assignments produced by Silph [26]. Each labeled example consists of the circuit graph (V,E), node features ϕ_S, and the target assignment $y := S^{\text{Silph}}(c)$, i.e., the optimal ILP-based assignment under Silph's cost model [26]. We use a cross-entropy loss over protocols at each node. To discourage invalid assignments, we add an auxiliary penalty term $\texttt{IA}(\theta,x)$ that scores how often the predicted protocol for a node would violate the constraints in §2.2.4. Concretely, $\texttt{IA}(\theta,x) = \sum_{i=1}^{n} S_\theta^{\text{SING}}(x_i) \cdot \gamma_i$, where $\gamma_i \in \{0,1\}^{|P|}$ marks which protocols are invalid for node i. The total supervised loss is

$$\mathcal{L}(\theta,x,y) = \alpha\,\texttt{CrossEntropy}(\theta,x,y) + (1-\alpha)\,\texttt{IA}(\theta,x),$$

with $\alpha \in [0,1]$. We study the effect of α in §8.4.

7.2.2 SING 2: Semi-supervised training with Silph [26]

In our second training regimen, we train our assignment model without supervised access to Silph's optimal assignment. Instead, the share assignment model S^{SING} is optimized to minimize the cost prediction C^{SING}. In this setup, C^{SING} is first trained to approximate Silph's assigned cost for the circuit-assignment $C^{\text{Silph}}(c,s)$ (cf. §6) and is then frozen for the subsequent semi-supervised training of S^{SING}.

For an unlabeled circuit c, we let the model propose an assignment $\hat{s} = S_\theta^{\text{SING}}(c)$, and we treat $C^{\text{SING}}(c,\hat{s})$ as a differentiable loss term. Intuitively, the model learns to output assignments that are cheap according to the cost prediction function. Since C^{SING} is only trained on valid assignments, this loss alone does not penalize invalid protocol combinations. We use α again to balance cost minimization and assignment validity, and retain the invalid assignment loss:

$$\mathcal{L}(\theta,x) = \alpha C^{\text{SING}}(c, S_\theta^{\text{SING}}(x)) + (1-\alpha)\,\texttt{IA}(\theta,x).$$

7.2.3 SING 3: Bootstrapped semi-supervised training

This training setup is identical to SING 2, except that the frozen cost model $C^{\text{SING}}(c,s)$ predicts the measured real-world runtimes C^{Runtime}. This enables our framework SING to train a share assignment model completely independently of any previously established MPC cost model, provided a high-quality cost predictor C^{Runtime} is available. We note that Silph's assignments are used to construct the cost predictors C^{Silph} and C^{Runtime} (cf. §6), ensuring a wide distribution of share assignments. We leave it to future work to construct hybrid cost predictors independent of Silph.

8 Evaluation

We evaluate SING's cost prediction model C^{SING} (§6) and share assignment model S^{SING} (§7). First, we compare both models to the state-of-the-art MPC share assignment framework Silph [26] (§8.2, §8.3, §8.5). Then, we investigate the cost of retraining our ML models (§8.6), which is of interest when retargeting our framework to new MPC configurations (e.g., different network settings, new protocols, or MPC implementations). Finally, we perform ablation studies to validate our training process (§8.7).

8.1 Implementation and Experimental Setup

We implement SING as a standalone pipeline that interfaces with existing MPC toolchains. We invoke Silph [26] to compile C code into hybrid circuits, obtain Silph's ILP-based share assignments and analytical cost estimates. We run circuits with ABY [31] for empirical runtimes and communication costs of assignments. We implement all learning components in PyTorch [4] and PyTorch Geometric [35]. The circuits are then parsed into graph objects with node features and protocol annotations, and split into training/validation/test sets (cf. §5.4.5) for training the cost prediction and share assignment models. We track and log experiments with standard tools (torcheval, torchmetrics, and Weights & Biases). Besides the core training code, our repository includes scripts for dataset generation (§5), running ABY [31], and end-to-end evaluation (§8). We performed all benchmarks on bare-metal hosts with AMD EPYC 9254 CPUs and 1 TB of RAM. The operating system was Arch Linux with kernel version 6.12.29. Model training was run on an NVIDIA RTX 6000 GPU. We simulate different network settings using scripts adapted from NEON [61]. The LAN setting has a throughput of 1 Gbit/s and a round-trip latency of 1 ms. The WAN throughput is 100 Mbit/s and latency is 100 ms. Our software stack consisted of Python 3.13.5, PyTorch [4] 2.6.0, and PyTorch Geometric [35] 2.6.1. We provide the learning rate, epoch count, and other hyperparameters for each model individually in §B, as these vary.

8.2 Cost Prediction

We evaluate our MPC cost prediction models for the two target functions (cf. §6): Silph's analytical cost model C^{Silph} [26] and empirical runtimes C^{Runtime}. For C^{Runtime}, we reduce measurement noise by averaging each circuit's execution time

Table 1: Analysis of cost predictors with respective targets, comparing cost models of SING and Silph [26]. C^{SING} is evaluated on the test split.

Ground truth	Predictor	R^2-score[1]	MSE[2]
C^{Runtime}	C^{Silph}	−95,611.734	16,870,668.0
C^{Silph}	C^{SING}	0.817	10,123,361.0
C^{Runtime}	C^{SING}	−1.290	3.7

[1] higher is better, range $(-\infty, 1]$
[2] Mean Squared Error, lower is better, range $[0, \infty)$

over 10 WAN runs. To bound dataset generation time, we exclude benchmarks with runtimes exceeding 60 s. To quantify the gap between both training regimes, we additionally assess how well C^{Silph} predicts real-world runtimes. The C^{Silph} dataset contains 46,788 data points in total, while C^{Runtime} contains 45,099 data points.

We evaluate prediction quality on the test split using the R^2-score and Mean Squared Error (MSE). The MSE measures regression error (lower is better), whereas R^2 captures how much variance in the ground truth is explained by the predictions (higher is better). Table 1 reports the results. Both metrics show that SING's real-world cost model C^{SING} approximates measured runtimes C^{Runtime} substantially better than Silph's analytical model C^{Silph}. This is expected: C^{Silph} is calibrated on microbenchmarks performed on a different system and intended to guide Silph's internal ILP, not to predict end-to-end MPC runtime precisely. At the same time, the additive structure of C^{Silph} makes it comparatively easy for C^{SING} to imitate, as reflected by the high R^2-score of 0.817 when predicting C^{Silph}. The corresponding MSE is large because it is not normalized: identical relative errors yield larger MSE values when the baseline magnitudes are larger, which is the case for C^{Silph}.

In contrast, C^{Runtime} consists of wall-clock runtimes in seconds and therefore typically exhibits smaller absolute values than C^{Silph}, resulting in a smaller MSE. However, the negative R^2-score indicates limited generalization in this setting. Empirically, we found that the performance of C^{SING} is sensitive to the quality and distribution of circuits, share assignments (e.g., different perturbations), and the measurement setup (LAN vs. WAN). This is expected, as real-world runtimes vary substantially across these factors.

We used both cost predictors C^{SING} as training surrogates for semi-supervised training. We analyze the suitability of C^{SING} as a loss function for semi-supervised training of the share assignment model S^{SING} in §8.5.

8.3 Share Assignment

In this section, we compare the share assignment runtime of SING with Silph [26]. More precisely, on the same hardware, we compare the CPU inference time of S^{SING} with the ILP runtime of Silph. Silph was run with the `empirical` cost model and the `smart_glp` selection scheme, i.e., all functions were inlined, and no graph partitioning was performed. This configuration yields the best output quality, albeit at the expense of slower share assignment. We exclude the overhead of other operations that contribute to the total cost of share assignment (e.g., parsing the circuit and loading the model parameters). The inference time of S^{SING} does not depend on whether the model was trained in a supervised or semi-supervised setting, as the model architecture is identical.

Table 2: Comparison of SING's share assignment times [ms] with Silph [26], sorted by Silph share assignment time. Silph CSS (call-site similarity) crashed for 5/9 circuits, denoted by an empty cell. For consistency, speedup is measured relative to Silph (without CSS). We report aggregate speedups as geometric means over per-circuit speedups.

Circuit	Silph [26]	+CSS [26]	SING (ours)	Speedup [×]
GCD	28	36	2	17.1
Gauss	112	174	1	95.2
DB Merge	283	181	2	170.0
Histogram	231,035		11	21,118.7
CryptoNets [39]	1,086,481		134	8,081.7
DB Join	1,143,312		19	60,546.2
Biomatch	1,259,105		146	8,636.9
k-means	1,479,436	498	19	76,696.5
MiniONN [70]	36,381,448		778	46,738.3
Geometric mean	-	-	-	3,544.7

Table 2 compares share assignment runtimes for S^{SING} and Silph's ILP. Across all benchmark circuits, SING computes share assignments substantially faster than Silph. The speedup grows with circuit size; for MiniONN [70], our largest circuit, SING achieves a speedup of 46,738 × over Silph. However, such speedups are only meaningful if assignment quality remains comparable; we evaluate this in §8.5.

8.4 Invalid Assignment Loss

The blended loss functions used to train our share assignment model S^{SING} are parameterized by $\alpha \in [0, 1]$, which controls the relative weight of the invalid-assignment loss (cf. §7.2). Setting $\alpha = 1$ removes the invalid-assignment loss, whereas $\alpha = 0$ removes the other term (cross-entropy in the supervised setting and C^{SING} in the semi-supervised setting).

In supervised training (SING 1, cf. §7.2.1), $\alpha = 0.9$ yielded the best results. With $\alpha = 0.1$, the cross-entropy term is down-weighted so strongly that the model fails to learn and assigns every operation to `yao` for the first 138 epochs, resulting in an

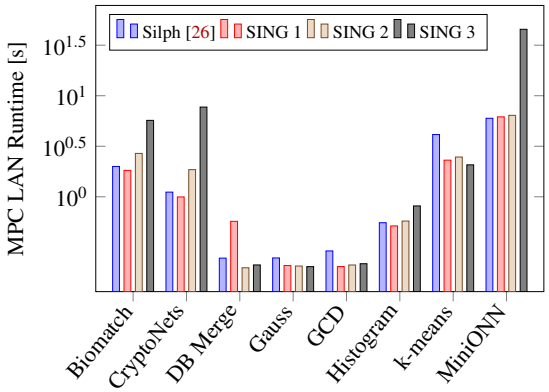


Figure 2: Comparison of resulting MPC LAN runtimes (s) for different share assignments. Detailed results in Table 6.

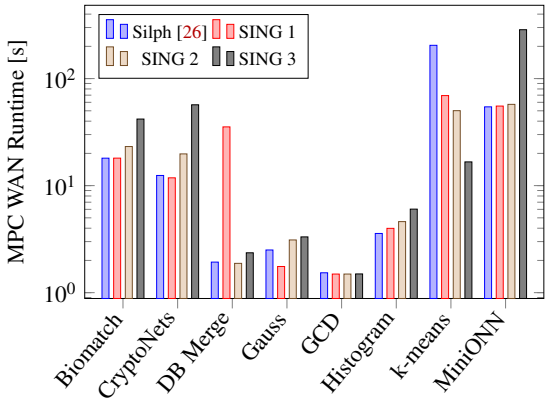


Figure 3: Comparison of resulting MPC WAN runtimes (s) for different share assignments. Detailed results in Table 7.

accuracy of 0.844 compared to 0.921 for $\alpha = 0.9$. Conversely, large α values increase the risk of invalid assignments: without penalizing invalid assignments (i.e., $\alpha = 1$), 280 share assignments in the test split are invalid after training.

For semi-supervised training (SING 2, cf. §7.2.2, and SING 3, cf. §7.2.3), we observed the same behavior when changing α. However, the optimal α depends on the scale of the C^{SING} term: if C^{SING} produces larger values, α must be smaller to prevent the invalid assignment loss from becoming negligible in the overall objective. We leave a full hyperparameter sweep over α to future work.

8.5 MPC Performance

In this section, we show that SING's share assignments S^{SING} achieve quality comparable to Silph [26], despite substantially faster inference (cf. §8.3). We evaluate our supervised and semi-supervised models, SING 1 (§7.2), SING 2 (§7.2.2), and SING 3 (§7.2.3), against share assignments produced by Silph using its `empirical` cost model and the `smart_glp` selection

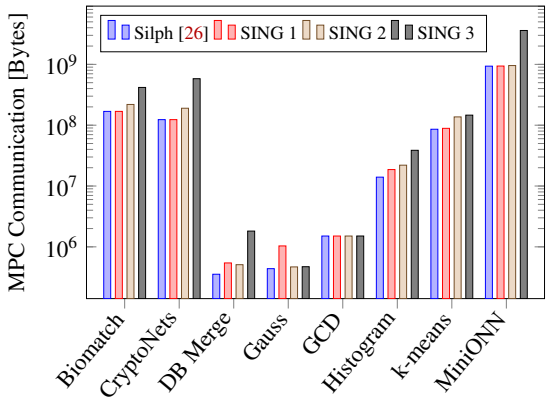


Figure 4: Resulting MPC communication (Bytes) for different share assignments; detailed results in Table 8.

scheme. This Silph configuration prioritizes optimality, but incurs higher share-assignment runtimes than other ones. The MPC runtimes exclude share-assignment times, which were compared independently in the previous subsection.

We report LAN runtimes in Fig. 2, WAN runtimes in Fig. 3, and communication volumes in Fig. 4. A detailed breakdown of all measured runtimes and communication volumes is provided in §B. For most benchmarks, the MPC runtime and communication achieved by share assignments generated by SING 1 and SING 2 are similar to Silph, excluding two outlier circuits. Due to its lower throughput and higher latency, the differences in runtime are more pronounced in the WAN setting.

Outliers Two circuits show significant runtime deviations:

1) *DB Merge* contains a division operation, which SING 1 assigns to `bool` instead of `yao`, increasing runtime by 17.3 × in WAN. This large impact is due to the circuit's small size.

2) *k-means*, where our models consistently outperform Silph. Silph assigns many `SELECT`, `STORE`, and other non-arithmetic operations to `bool` and `yao` alternatingly, while our models consistently assign `yao`. Silph's assignment results in additional conversions, thus increasing the number of communication rounds, demonstrating the deviation of the simplified cost model C^{Silph} from real-world runtimes.

Fully bootstrapped model SING 3 (cf. §7.2.3) was obtained by training a share assignment model S^{SING} without any ground truth derived from Silph and relies solely on benchmarked MPC runtimes. Due to the high variance of MPC runtimes, this is a more complex task with higher requirements on the dataset quality. SING 3 performs on par with our other models for most circuits. Exceptions are Biomatch, CryptoNets, and MiniONN , where SING 3 assigns all additions to `yao` instead of `arith` leading to additional protocol conversion overhead. Our benchmarks show that C^{SING}

and S^{SING} can partially capture how share assignments affect MPC performance *without external guidance*, despite C^{SING} showing limited generalization on unseen circuits during training (cf. §8.2). At the same time, SING 3's performance required multiple iterations of hyperparameter tuning and dataset refinement; improving training robustness remains important future work.

8.6 Model Training Performance

An important aspect of our framework's practicality is the effort and resources required to retrain our share assignment models from scratch. This matters whenever we need to adapt to a new MPC deployment scenario, e.g., due to changes in hardware or network characteristics, the use of different MPC protocols, or a different underlying MPC implementation.

Table 3 shows that training our models is fast in practice. All experiments were run on the hardware described in §8.1. Since training time scales with dataset size, training C^{SING} takes multiple orders of magnitude longer than training S^{SING}. Semi-supervised training of S^{SING} is slightly slower than supervised training because evaluating the C^{SING}-based loss is more expensive than `CrossEntropy`. Note that training time is independent of the specific training regime, since we use the same parameters throughout.

8.7 Ablation

Next, we conduct an ablation study to verify whether our ML models utilize the expected features. For both cost prediction and share assignment, we hypothesize that C^{SING} and S^{SING} leverage both *circuit structure* and *operation types*. We ablate each feature group in turn and measure the performance drop. If performance is largely unchanged, the model is not relying on the removed information. In each case, we keep the architecture fixed and train with the same hyperparameters as in the non-ablated setting. For semi supervised training of S^{SING}, we use the C^{SING}-based loss, where C^{SING} is trained on the standard non ablated dataset. We report our detailed ablation results in Table 5 (in §B) and discuss the results here.

To test whether our models leverage **circuit structure**, we remove all edges during training and evaluate our model's performance on the original, unmodified dataset. Thus, the model predicts costs and infers share assignments solely based on the circuit's operation types. We found that removing edges does not significantly impact the accuracy. Consequently, the operation types carry most of the information for both cost prediction and share assignment. However, the average error increases to more than twice the baseline without ablation. This is expected because estimating share-conversion overheads requires edge information.

Looking at individual circuits provides a tangible example of the effects of removing edges: Listing 1 shows a synthetic benchmark circuit from the Silph [26] code base. It consists of a chain of alternating `ADD` and `MUX` operations. Without any edge information, S^{SING} will assign all `ADD`s to `arith` and all `MUX`s to `yao`, resulting in a LAN runtime of 228.93 s due to numerous protocol conversions and communication rounds. However, when trained with circuit structure, S^{SING} will assign all operations to `yao`, resulting in only 2.73 s runtime. This suggests that producing a *good-enough* share assignment is relatively simple and can be achieved from operation types alone, without the circuit structure. Moving beyond this baseline to obtain near-optimal assignments that explicitly account for data dependencies and circuit data flow is substantially more challenging. S^{SING} is able to capture these structural effects and produce such optimized assignments.

Table 3: Training runtime for our models.

Model	Dataset size	Epochs	Training [s]
C^{SING}	46,788	150	1,636
S^{SING} supervised	604	200	25
S^{SING} semi-supervised	604	200	37

Listing 1: Benchmark from Silph [26].

```
int main(__attribute__((private(0))) int a,
         __attribute__((private(1))) int b){
    int acc = 0;
    for (int i = 0; i < 75000; i++) {
        if (a > b) { acc += 1; } }
    return acc; }
```

To ablate **operation types**, we set the node features ϕ_C and ϕ_S to zero during training. This preserves circuit structure, but removes the `OneHot` operation encoding. This ablation degrades performance across all metrics. For S^{SING}, removing operation type features leads to invalid assignments. Since C^{Silph} assigns no cost to invalid assignments, these results are not comparable and therefore omitted.

9 Conclusion and Future Work

We presented SING, the first ML-based framework for MPC cost prediction and share assignment. It is generic and, in principle, extensible to new deployment settings and protocol suites. Our evaluation shows that, for two of the three training regimes, SING matches Silph's [26] share-assignment quality while running substantially faster. In the fully bootstrapped regime (i.e., SING 3), we train the share-assignment model directly on real-world runtimes. We found this approach to be brittle and to generalize poorly, as reflected by its negative R^2 value; it also required extensive hyperparameter tuning and dataset refinement. We attribute this primarily to the poor quality of the runtime cost surrogate, since semi-supervised training worked well with a simpler cost model (i.e., SING 2). We release our models, dataset, and code as artifacts.

We highlight four areas for future work: 1) Designing a larger and more diverse dataset spanning additional circuit and protocol mixes to improve training robustness, 2) Integrating more tightly with Silph [26], such that S^{SING} can provide high-quality warm starts to accelerate ILP solving and for better usability, 3) Using SING for the hybrid compilation of state-of-the-art protocol suites, such as [38], and 4) Exploring alternative learning paradigms. Reinforcement learning can cast the assignment as a sequential decision process, and generative models can synthesize additional circuit-like graphs to reduce distribution shift [16, 41]. Likewise, alternative graph architectures beyond GCN [59], and GraphSAGE [45], such as graph transformers [82], may improve generalization.

Acknowledgments

We warmly thank Marco Chilese for the early discussions that helped inspire this project. We also thank our shepherd and reviewers for their support and inputs. This project received funding from the ERC under the EU's research and innovation programs Horizon Europe (PRIVTOOLS/101124778) and Horizon 2020 (PSOTI/850990). It was co-funded by the DFG within SFB 1119 CROSSING/236615297 and Germany's excellence strategy EXC-3057, and supported by BMFTR and HMWK within ATHENE.

Ethical Considerations

This section details our stakeholder-based ethics analysis, following the conference ethics guidelines and the ethical principles in the Menlo report[2].

Stakeholders We identify MPC researchers, developers of MPC implementations, and users of MPC systems as the primary stakeholders directly impacted by our research. Overall, our results benefit these groups by drastically increasing the efficiency of MPC protocol assignments. At the same time, improving the concrete efficiency of MPC as a privacy-enhancing technology also makes MPC more accessible to malicious actors who might seek to misuse MPC.

In a broader sense, we view society at large as an indirectly affected stakeholder. As MPC protocols are increasingly deployed in real-world scenarios, society at large benefits from reduced computational waste and power consumption. Furthermore, by improving the concrete efficiency of these systems, SING makes MPC, as a privacy-enhancing technology, more accessible to a broader class of users who may have previously been limited by its high overheads.

Data Sourcing and Institutional Oversight In accordance with the Menlo Report's principle of *Respect for Persons*, we confirm that this research did not involve human subjects; as such, no Institutional Review or Ethics Board approval was required. The dataset used to train the SING Networks consists of MPC circuits derived from established open-source benchmarks (e.g., Silph [26] and ABY [31]) and synthetically generated graphs. All performance metrics, such as runtime and communication costs, were collected in isolated environments using automated scripts. From a privacy perspective, our system for optimizing hybrid MPC protocols is harmless, as it processes no personal data and does not have access to any: it processes abstract circuit descriptions and generates C code using generative AI (cf. §5.3). Hence, our research upholds the ethical principle of *Compliance*.

Transparency and Reproduciblity To support the conference's commitment to open science and the Menlo principle of *Transparency*, we provide our models, the circuit dataset, and the full SING framework as reproducible artifacts. The research team considered the ethics of this work, believes the research was conducted ethically, and confirms that all next steps, including the public release of code and data, follow responsible disclosure and open-science practices. We further recognize that *Reproducibility* is one of the cornerstones of scientific integrity, especially in machine learning. To uphold this ethical standard of (AI) research, we place great emphasis on reproducibility, ensuring that our technical claims are substantiated by auditable, repeatable workflows and guidelines for reproducing our work (cf. §B and the Open Science section below). The use of generative AI in editing, as well as in generating synthetic data, is fully disclosed in §A and described (cf. §5.3, §B.1).

Framework Integrity and Beneficence Following the principle of *Beneficence*, our goal is to maximize the utility of MPC by making it more efficient for the security community, without compromising its security. We emphasize that SING purely optimizes the mapping of computations to hybrid MPC protocols (cf. §6, §7); it does not modify the underlying cryptographic primitives, thereby maintaining the original security guarantees of the underlying MPC protocols. The security of any (valid) assignment directly results from standard secure composition theorems.

Potential Harms Analysis Delegating protocol assignment to ML may expose the *risk of suboptimal or invalid assignments*, leading to execution failures in deployment scenarios. We mitigate this by integrating physical and cryptographic constraints directly into our training process via a blended loss function (cf. §8.4), ensuring predictions respect protocol requirements. While GNNs can be opaque, the cryptographic integrity of the MPC execution remains independent of the ML-driven selection, preventing the introduction of new attack surfaces. To address potential automation bias and training data bias, we employed graph-level data augmen-

[2] https://www.dhs.gov/sites/default/files/publications/CSD-MenloPrinciplesCORE-20120803_1.pdf

tation and synthetic circuit generation to ensure the model learns generalized structural features rather than overfitting to specific benchmarks. Nonetheless, we strongly advise MPC practitioners and developers to carefully test their systems and strictly comply with applicable laws and regulations.

We also note that any efficiency improvement in MPC, such as our work, makes the technology itself more prone to dual-use if used by malicious actors to conceal unlawful activities. Releasing all our research artifacts publicly in a reproducible fashion may pronounce this issue further; yet, we strongly believe that the massive scalability gains of our framework represent a significant net benefit to the MPC community (including researchers, developers, and application users) and our society, which we find to outweigh the marginal and general risks of MPC misuse.

Decision to Proceed and Publish The authors attest that they have read the ethics considerations discussed in the conference call for papers and the detailed submission instructions. We believe that our research was conducted ethically and that our next steps, including the public release of the SING framework and its components, are designed to benefit the MPC community and broader security research community, while minimizing the potential for misuse.

Open Science

We publish the full code for our work and our circuit dataset on Zenodo[3]. The README inside the artifact provides extensive guidance on setting up the training and evaluation pipeline. We recommend setting up a Python virtual environment in which to install all dependencies. Setting up the initial environment works as follows:

```
python -m venv venv
source venv/bin/activate

pip install torch torchvision torchaudio
pip install -r requirements.txt

tar xf dataset-excerpt.tar.gz
tar xf dataset-excerpt-c.tar.gz
```

The installation of PyTorch varies based on the computation platform (e.g., CUDA). We refer to the PyTorch documentation[4] for more details.

Once the environment and the dataset are set up, we can train the supervised share assignment model S_{SING}.

```
python generate_dataset_split.py
mkdir -p dataset
ln -s dataset-excerpt dataset/raw
python train.py
```

[3] https://doi.org/10.5281/zenodo.20347719
[4] https://pytorch.org/get-started/locally/

The trained model can then be evaluated, including benchmarking MPC runtime and communication of share assignments produced by it. The repository's README describes this process in detail, together with instructions for training and evaluating our cost prediction model C_{SING}. The model checkpoints used for our evaluation are provided in a subdirectory. Additionally, the repository contains all other code not directly involved in training and evaluation (e.g., dataset processing and LLM circuit generation).

References

[1] Cosku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. Viaduct: an extensible, optimizing compiler for secure distributed programs. In *PLDI*, 2021.

[2] Abdelrahaman Aly et al. SCALE–MAMBA v1.14 : Documentation. https://nigelsmart.github.io/SCALE/Documentation-SCALE.pdf, 2021.

[3] Sean Eron Anderson. Bit twiddling hacks. https://graphics.stanford.edu/~seander/bithacks.html, 2005.

[4] Jason Ansel et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *ASPLOS*, 2024.

[5] Apple Support. Password monitoring. https://support.apple.com/guide/security/password-monitoring-sec78e79fc3b/web, 2025.

[6] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

[7] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *STOC*, 1990.

[8] Sebastian Becker et al. Carbyne stack. https://github.com/carbynestack/carbynestack, 2021.

[9] Mariya Georgieva Belorgey et al. Manticore: A framework for efficient multiparty computation supporting real number and boolean arithmetic. *J. Cryptol.*, 2023.

[10] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: a system for secure multi-party computation. In *CCS*, 2008.

[11] Christopher M. Bishop. Training with noise is equivalent to tikhonov regularization. *Neural Comput.*, 1995.

[12] Jannis Blüml, Moritz Huppert, Nora Khayata, Joachim Schmidt, and Thomas Schneider. SING: Improving the efficiency of MPC protocol assignment using graph neural networks. In *35th USENIX Security Symposium (USENIX Security 26)*, 2026.

[13] Dan Bogdanov, Peeter Laud, and Jaak Randmets. Domain-polymorphic language for privacy-preserving applications. In *PETShop, CCS*, 2013.

[14] Peter Bogetoft, Ivan Damgård, Thomas P. Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In *FC*, 2006.

[15] Peter Bogetoft et al. Secure multiparty computation goes live. In *FC*, 2009.

[16] Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. Netgan: Generating graphs via random walks. In *ICML*, 2018.

[17] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. MOTION - A framework for mixed-protocol multi-party computation. *ACM Trans. Priv. Secur.*, 2022.

[18] Lennart Braun, Moritz Huppert, Nora Khayata, Thomas Schneider, and Oleksandr Tkachenko. FUSE - flexible file format and intermediate representation for secure multi-party computation. In *AsiaCCS*, 2023.

[19] Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. Privacy-preserving remote diagnostics. In *CCS*, 2007.

[20] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Process. Mag.*, 2017.

[21] Andreas Brüggemann, Robin Hundt, Thomas Schneider, Ajith Suresh, and Hossein Yalame. FLUTE: fast and secure lookup table evaluations. In *S&P*, 2023.

[22] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. Hycc: Compilation of hybrid protocols for practical secure computation. In *CCS*, 2018.

[23] Niklas Büscher, Andreas Holzer, Alina Weber, and Stefan Katzenbeisser. Compiling low depth circuits for practical secure computation. In *ESORICS*, 2016.

[24] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. Ezpc: Programmable and efficient secure two-party computation for machine learning. In *EuroS&P*, 2019.

[25] ChatGPT. Chatgpt. Accessed: 2025-01-29. URL: https://chatgpt.com/.

[26] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Wenting Zheng. Silph: A framework for scalable and accurate generation of hybrid MPC protocols. In *S&P*, 2023.

[27] Lei Chen et al. Large circuit models: opportunities and challenges. *Sci. China Inf. Sci.*, 2024.

[28] 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. *IACR Cryptol. ePrint Arch.*, 2025. URL: https://ia.cr/2025/1315.

[29] DeepSeek. Deepseek. Accessed: 2025-01-27. URL: https://chat.deepseek.com/.

[30] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. Automated synthesis of optimized circuits for secure computation. In *CCS*, 2015.

[31] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[32] Henri Dohmen, Robin Hundt, Nora Khayata, and Thomas Schneider. SEEC: memory safety meets efficiency in secure two-party computation. In *AsiaCCS*, 2025.

[33] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO*, 2020.

[34] 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 *EuroS&P*, 2022.

[35] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric, 2019. arXiv:1903.02428.

[36] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: an ANSI C compiler for secure two-party computations. In *CC*, 2014.

[37] Radhika Garg and Xiao Wang. Smaug: Modular augmentation of LLVM for MPC. In *S&P*, 2025.

[38] Radhika Garg, Kang Yang, Jonathan Katz, and Xiao Wang. Scalable mixed-mode MPC. In *S&P*, 2024.

[39] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML*, 2016.

[40] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC*, 1987.

[41] Ian J. Goodfellow et al. Generative adversarial nets. In *NeurIPS*, 2014.

[42] Daniel Günther, Ágnes Kiss, and Thomas Schneider. More efficient universal circuit constructions. In *ASIACRYPT*, 2017.

[43] Kanav Gupta, Neha Jawalkar, Ananta Mukherjee, Nishanth Chandran, Divya Gupta, Ashish Panwar, and Rahul Sharma. SIGMA: secure GPT inference with function secret sharing. In *Proc. Priv. Enhancing Technol.*, 2024.

[44] William L. Hamilton. *Graph Representation Learning*. Springer Cham, 2020. ISBN: 978-3-031-00460-5.

[45] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NeurIPS*, 2017.

[46] Tim Heldmann, Thomas Schneider, Oleksandr Tkachenko, Christian Weinert, and Hossein Yalame. Llvm-based circuit compilation for practical secure computation. In *ACNS*, 2021.

[47] Brett Hemenway, Steve Lu, Rafail Ostrovsky, and William Welser IV. High-precision secure computation of satellite collision probabilities. In *SCN*, 2016.

[48] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In *CCS*, 2010.

[49] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012. arXiv:1207.0580.

[50] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.

[51] Siam U. Hussain, Baiyu Li, Farinaz Koushanfar, and Rosario Cammarota. Tinygarble2: Smart, efficient, and scalable yao's garble circuit. In *PPMLP@CCS*, 2020.

[52] Muhammad Ishaq, Ana L. Milanova, and Vassilis Zikas. Efficient MPC via program analysis: A framework for efficient optimal mixing. In *CCS*, 2019.

[53] Daniel Cabarcas Jaramillo, Hernán Darío Vanegas Madrigal, Daniel Escudero, and Fernando Alberto Morales Jauregui. Privacy-preserving training of support vector machines via secure multiparty computation. *ACM Trans. Priv. Secur.*, 2025.

[54] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. Orca: Fss-based secure training and inference with gpus. In *S&P*, 2024.

[55] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*, 2020.

[56] Marcel Keller and Ke Sun. Secure quantized training for deep learning. In *ICML*, 2022.

[57] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. Automatic protocol selection in secure two-party computations. In *ACNS*, 2014.

[58] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

[59] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

[60] Ágnes Kiss and Thomas Schneider. Valiant's universal circuit is practical. In *EUROCRYPT*, 2016.

[61] Andreas Klinger, Vincent Ehrmanntraut, and Ulrike Meyer. Estimating the runtime and global network traffic of SMPC protocols. In *CODASPY*, 2024.

[62] Denis Kocetkov et al. The stack: 3 TB of permissively licensed source code, 2023.

[63] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *CANS*, 2009.

[64] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, 2008.

[65] Vladimir Kolesnikov and Thomas Schneider. A practical universal circuit construction and secure evaluation of private functions. In *FC*, 2008.

[66] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *(CGO*, 2004.

[67] Benjamin Levy, Muhammad Ishaq, Benjamin Sherman, Lindsey Kennard, Ana L. Milanova, and Vassilis Zikas. COMBINE: compilation and backend-independent vectorization for multi-party computation. In *CCS*, 2023.

[68] Fang Liu, Dong Wang, and Tian Yan. Some examples of privacy-preserving sharing of covid-19 pandemic data with statistical utility evaluation. *BMC Medical Research Methodology*, 2023. doi:10.1186/s12874-023-01927-3.

[69] Hanlin Liu, Yu Yu, Shuoyao Zhao, Jiang Zhang, Wenling Liu, and Zhenkai Hu. Pushing the limits of valiant's universal circuits: Simpler, tighter and more compact. In *CRYPTO*, 2021.

[70] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. In *CCS*, 2017.

[71] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *NeurIPS*, 2023.

[72] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Outperforming GPT-3.5 in design RTL generation with our open-source dataset and lightweight solution, 2023. arXiv:2312.08617.

[73] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *USENIX*, 2004.

[74] OpenSSL. Openssl library. https://openssl-library.org/, 2025.

[75] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. In *S&P*, 2022.

[76] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[77] Qi Pang, Jinhao Zhu, Helen Möllering, Wenting Zheng, and Thomas Schneider. BOLT: privacy-preserving, accurate and efficient inference for transformers. In *S&P*, 2024.

[78] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: improved mixed-protocol secure two-party computation. In *USENIX*, 2021.

[79] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. Syncirc: Efficient synthesis of depth-optimized circuits for secure computation. In *HOST*, 2021.

[80] Erman Pattuk, Murat Kantarcioglu, Huseyin Ulusoy, and Bradley A. Malin. Cheapsmc: A framework to minimize secure multiparty computation cost in the cloud. In *DBSec*, 2016.

[81] Michael O. Rabin. How to exchange secrets with oblivious transfer. Technical report, Harvard University, 1981. URL: https://ia.cr/2005/187.

[82] Ladislav Rampásek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer. In *NeurIPS*, 2022.

[83] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification. In *ICLR*, 2020.

[84] Emanuele Rossi, Bertrand Charpentier, Francesco Di Giovanni, Fabrizio Frasca, Stephan Günnemann, and Michael M. Bronstein. Edge directionality improves learning on heterophilic graphs. In *Learning on Graphs Conference*, 2023.

[85] Mike Rosulek and Lawrence Roy. Three halves make a whole? beating the half-gates lower bound for garbled circuits. In *CRYPTO*, 2021.

[86] Thomas Schneider and Michael Zohner. GMW vs. yao? efficient secure two-party computation with low depth circuits. In *FC*, 2013.

[87] Axel Schröpfer and Florian Kerschbaum. Forecasting run-times of secure two-party computation. In *International Conference on Quantitative Evaluation of Systems (QEST)*, 2011.

[88] Axel Schröpfer, Florian Kerschbaum, and Günter Müller. L1 - an intermediate language for mixed-protocol secure computation. In *COMPSAC*, 2011.

[89] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *J. Big Data*, 2019.

[90] SINE Foundation. Maliciously-secure multi-party computation (mpc) engine using authenticated garbling. https://github.com/sine-fdn/polytune, 2023.

[91] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *S&P*, 2015.

[92] European Union. General Data Protection Regulation (GDPR). https://gdpr-info.eu/, 2016.

[93] Leslie G. Valiant. Universal circuits (preliminary report). In *STOC*, 1976.

[94] Marie Beth van Egmond, Vincent Dunning, Stefan van den Berg, Thomas Rooijakkers, Alex Sangers, Ton

Poppe, and Jan Veldsink. Privacy-preserving anti-money laundering using secure multi-party computation. In *FC*, 2024.

[95] Ashish Vaswani et al. Attention is all you need. In *NeurIPS*, 2017.

[96] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. Emp toolkit. https://github.com/emp-toolkit, 2016.

[97] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *CCS*, 2017.

[98] Yuxiang Wei et al. Selfcodealign: Self-alignment for code generation, 2024.

[99] Yutong Wu et al. Inversecoder: Self-improving instruction-tuned code llms with inverse-instruct. In *AAAI*, 2025.

[100] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, 1986.

[101] Yuning You, Tianlong Chen, Yongduo Sui, Ting Chen, Zhangyang Wang, and Yang Shen. Graph contrastive learning with augmentations. In *NeurIPS*, 2020.

[102] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT*, 2015.

[103] Tong Zhao, Wei Jin, Yozen Liu, Yingheng Wang, Gang Liu, Stephan Günnemann, Neil Shah, and Meng Jiang. Graph data augmentation for graph machine learning: A survey. *IEEE Data Eng. Bull.*, 2023.

A Generative AI Usage

As described in §5.3, we have used large language models (LLMs) to generate MPC-compatible C code, which makes up a significant part of our dataset for training SING. We have prompted ChatGPT, DeepSeek, and locally run open-source LLMs via `ollama`. All open-source LLMs we used are: codegemma, codellama, deepseek-coder, dolphin-mistral, dolphin3, gemma2, gemma3, llama3.1, llama3.2, mistral-nemo, mistral, phi3.5, qwen2.5-coder, qwen2.5, stable-code, stablelm2, starcoder2, and starcoder.

As mentioned in §5.3, from an initial 8557 generated programs, only 450 survived filtering. Our filtering discarded samples that failed to compile with Silph [26] or were duplicates. This filtering mechanism ensures that the resulting circuits are useful and compileable programs, thereby verifying the output of the used LLMs. We include our LLM prompts in our anonymous repository. Furthermore, this paper was edited for grammar using Grammarly.

Table 4: Hyperparameters of cost prediction model C^{SING} and share assignment model S^{SING}.

Hyperparameter	C^{SING}	SING 1	SING 2	SING 3
Epoch count	150	200	200	200
Batch size	32	32	32	32
Optimizer	Adam [58]	Adam	Adam	Adam
Dropout rate [49]	0.5	0.5	0.5	0.5
Learning rate	0.001	0.01	0.001	0.001
Weight decay	5×10^{-4}	5×10^{-4}	5×10^{-4}	5×10^{-4}
Loss blending factor α	-	0.9	0.1	0.99

B SING Details

We report the **hyperparameters** of our cost prediction models C^{SING} and share assignment models S^{SING} in Table 4. We report *detailed* ablation results (cf. §8.7) in Table 5. We further report *more detailed* LAN runtimes from our evaluation (cf. §8) in Table 6, WAN runtimes in Table 7, and communication in Table 8.

Table 5: Ablation results for our MPC cost prediction and MPC share assignment models.

Model	R^2 [1]	MSE [2]	Accuracy	C^{Silph} cost	Invalid
C^{SING}	0.992	1,727,542.6	-	-	-
no edges	0.984	3,470,813.5	-	-	-
no features	0.969	6,759,841.0	-	-	-
S^{SING} sup.	-	-	0.98	23,468.5	0
no edges	-	-	0.97	23,810.4	0
no features	-	-	0.36	185,385.7	0
S^{SING} semi-sup.	-	-	-	22,733.6	0
no edges	-	-	-	23,684.2	0
no features	-	-	-	-	46,722

[1] higher is better, range $(-\infty, 1]$
[2] lower is better, range $[0, \infty)$

B.1 Collecting MPC Benchmark Circuits

Our dataset only includes circuits from two MPC frameworks: Silph [26] and LLVM for MPC [46]. There are many more MPC frameworks with example circuits that could have theoretically been of interest for our dataset, although circuits are often ported from earlier MPC frameworks for evaluation.

LLM Prompts We utilized 6 different prompts for generating training data (cf. §5.3), which are available in our repository. In general, the more we specified the constraints of MPC circuits, the better the results were. Adding examples of what is expected also had a positive impact. However, we observed that LLMs will reproduce provided examples without modifications, even if the prompt specifies that these are only examples. For instance, in some prompts we specify an

Table 6: Comparison of MPC LAN runtimes in seconds with Silph [26]. Δ denotes the difference with respect to Silph. Aggregate differences are computed as geometric means over per-circuit runtime differences.

Circuit	Silph [26]	SING 1 (supervised)	Δ [%]	SING 2 (minimize C^{Silph})	Δ [%]	SING 3 (minimize C^{Runtime})	Δ [%]
Biomatch	2.00	1.83	−8.59	2.69	+34.56	5.70	+185.64
CryptoNets [39]	1.11	1.00	−10.39	1.86	+66.90	7.72	+594.19
DB Merge	0.25	0.57	+131.26	0.20	−19.72	0.21	−14.39
Gauss	0.25	0.21	−15.92	0.21	−16.85	0.20	−17.97
GCD	0.29	0.20	−29.93	0.21	−27.37	0.22	−25.25
Histogram	0.55	0.51	−7.05	0.58	+3.84	0.81	+46.61
k-means	4.13	2.30	−44.22	2.47	−40.10	2.07	−49.87
MiniONN [70]	5.98	6.18	+3.28	6.39	+6.86	45.41	+659.06
Geometric mean	-	-	−6.23	-	−3.96	-	+66.15

Table 7: Comparison of MPC WAN runtimes in seconds with Silph [26]. Δ denotes the difference with respect to Silph. Aggregate differences are computed as geometric means over per-circuit runtime differences.

Circuit	Silph [26]	SING 1 (supervised)	Δ [%]	SING 2 (minimize C^{Silph})	Δ [%]	SING 3 (minimize C^{Runtime})	Δ [%]
Biomatch	18.05	18.07	+0.08	23.18	+28.41	41.87	+131.95
CryptoNets [39]	12.44	11.82	−4.95	19.78	+59.03	56.90	+357.40
DB Merge	1.93	35.38	+1733.38	1.88	−2.66	2.36	+22.18
Gauss	2.50	1.76	−29.88	3.11	+24.11	3.32	+32.52
GCD	1.53	1.49	−2.67	1.49	−2.66	1.50	−2.43
Histogram	3.57	3.99	+11.64	4.61	+29.04	6.03	+68.79
k-means	204.55	69.36	−66.09	50.14	−75.49	16.66	−91.86
MiniONN [70]	54.44	55.36	+1.69	57.43	+5.49	286.09	+425.52
Geometric mean	-	-	+20.96	-	−2.73	-	+36.58

Table 8: Comparison of MPC communication amount in MBytes with Silph [26]. Δ denotes the overhead of SING over Silph.

Circuit	Silph [26]	SING 1 (supervised)	Δ [%]	SING 2 (minimize C^{Silph})	Δ [%]	SING 3 (minimize C^{Runtime})	Δ [%]
Biomatch	168.63	168.63	+0.00	219.22	+30.01	417.60	+147.65
CryptoNets [39]	123.41	123.44	+0.03	190.03	+53.99	581.18	+370.95
DB Merge	0.35	0.54	+53.33	0.51	+43.53	1.82	+412.29
Gauss	0.44	1.04	+136.45	0.47	+6.85	0.47	+7.68
GCD	1.51	1.51	+0.00	1.51	+0.00	1.51	+0.10
Histogram	13.99	18.65	+33.29	22.02	+57.35	38.66	+176.34
k-means	85.81	88.77	+3.45	136.87	+59.50	146.31	+70.50
MiniONN [70]	934.40	938.54	+0.44	956.16	+2.33	3605.92	+285.91

example function signature, which contains arrays of size 20, resulting in many output circuits having arrays of size 20 in their function signature, even though we want varying sizes.

Train-val-test Splitting In many settings, randomly splitting a dataset drawn from a single underlying distribution is sufficient. Our dataset, however, comprises circuits of widely varying sizes and heterogeneous share assignment distributions. We found that a random but deterministic split based on the unique SHA-256 hashes of the input C files does not yield balanced splits. The root cause is that a small number of very large benchmark circuits contain a substantial fraction of all nodes. Their share assignment distributions may not match the dataset-wide distribution, so placing even a single large circuit into a split can noticeably skew that split. To mitigate this, we implement a custom splitting strategy that accounts for share assignment distributions (cf. §5.4).