CostCO: An automatic cost modeling framework for secure multi-party computation

Vivian Fang
UC Berkeley
vivian@eecs.berkeley.edu

Lloyd Brown
UC Berkeley
lloydbrown@berkeley.edu

William Lin
UC Berkeley
will.lin@berkeley.edu

Wenting Zheng
CMU
wenting@cmu.edu

Aurojit Panda
NYU
apanda@cs.nyu.edu

Raluca Ada Popa
UC Berkeley
raluca@eecs.berkeley.edu

2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P) | 978-1-6654-1614-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/EUROSP53844.2022.00017

***Abstract*—The last decade has seen an explosion in the number of new secure multi-party computation (MPC) protocols that enable collaborative computation on sensitive data. No single MPC protocol is optimal for all types of computation. As a result, researchers have created *hybrid-protocol* compilers that translate a program into a hybrid protocol that mixes different MPC protocols. Hybrid-protocol compilers crucially rely on accurate cost models, which are hand-written by the compilers' developers, to choose the correct schedule of protocols.**

In this paper, we propose CostCO, the first *automatic* MPC cost modeling framework. CostCO develops a novel API to interface with a variety of MPC protocols, and leverages domain-specific properties of MPC in order to enable efficient and automatic cost-model generation for a wide range of MPC protocols. CostCO employs a two-phase experiment design to efficiently synthesize cost models of the MPC protocol's runtime as well as its memory and network usage. We verify CostCO's modeling accuracy for several full circuits, characterize the engineering effort required to port existing MPC protocols, and demonstrate how hybrid-protocol compilers can leverage CostCO's cost models.

1. Introduction

Secure collaborative computation [12], [17], [32], [45], [47], [59], [62], is an increasingly popular paradigm where multiple organizations with sensitive data run an analysis over their *aggregate* data, without revealing their individual sensitive data to each other. Computing on multiple parties' data is both advantageous and necessary in many cases ranging from finance to healthcare. For example, money laundering—where criminals transfer assets across financial institutions to mask their activities—is illegal in most jurisdictions and banks are required to detect and report such activity. However, detecting money-laundering is challenging because it requires banks to share sensitive customer transaction data with each other [57], and they are unwilling to do so because of business competition.

Over the past three decades, researchers have made impressive progress towards a cryptographic approach to this problem: *secure multi-party computation* (MPC) [4], [14], [30], [35], [36], [39], [48], [64], [66]. At a high level, MPC allows n parties $p_1, \ldots, p_n$ with corresponding inputs $x_1, \ldots, x_n$ to learn the output of a public function $f(x_1, \ldots, x_n)$ without revealing each party's x_i to other parties. Due to these efforts, MPC is now efficient enough for several real-world use cases [12], [17], [32], [45], [47], [59], [62]. Because there is no single MPC protocol that wins for all workloads, researchers have begun to design a number of *hybrid protocols* [7], [21], [23], [25], [34], [41], [43], [54] that combine different MPC protocols to bring orders of magnitude performance improvement over using just one MPC protocol to run the entire workload.

Consequently, *hybrid-protocol compilers* [11], [13], [33], [50] emerged in an effort to automate the manual process of optimally combining different MPC protocols for a given application. Fig. 1 depicts the standard workflow of a hybrid-protocol compiler. In order to construct a hybrid protocol for a given program, the compiler partitions the program and uses a cost model to select the MPC protocol to run on each partition. The resulting hybrid protocol can then be executed by the parties who want to securely compute the program.

A hybrid-protocol compiler crucially depends on its *cost model*, which it queries to determine the concrete costs of different options for protocol assignment. The main problem with existing cost models is the manual effort needed to derive them. Multiple issues arise as a result:

- *Burdensome effort.* Deriving a cost model for a specific MPC protocol requires a deep understanding of its runtime complexity. For example, EzPC [13] relies on distilling the properties of ABY [15] into hard coded heuristics specialized to a deployment and Kerschbaum, et. al [37], [56] manually derive cost models for every building block in their two supported MPC protocols. Others [11], [33], [50] set up experiments by hand and fix parameters like the data size.
- *Lack of extensibility.* Existing compilers base their cost models on certain assumptions about the deployment environment and the supported MPC protocols. EzPC [13] relies on rigid heuristics while newer compilers [11], [33], [50] use a cost model specific to the set of protocols they initially support. Both approaches make it difficult to integrate new MPC protocols into the compiler; new heuristics need to

DOI 10.1109/EuroSP53844.2022.00017

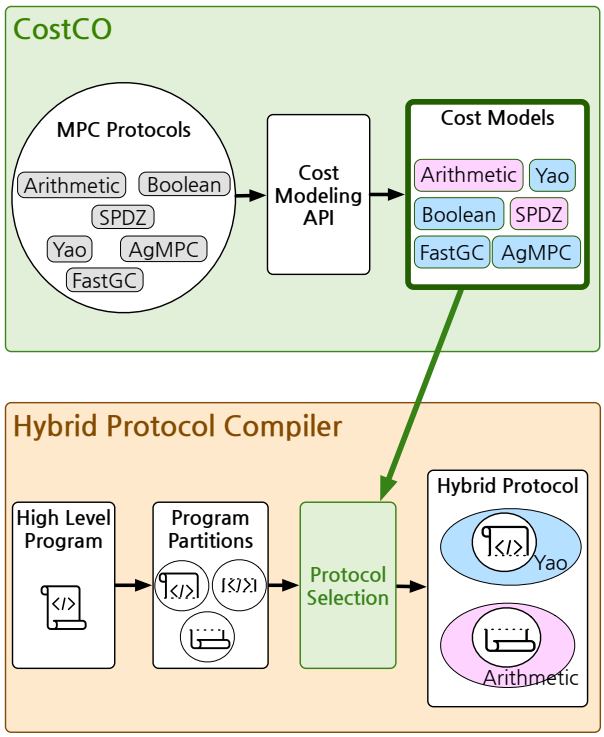

Figure 1. The standard workflow of hybrid-protocol compilers is drawn on the bottom (orange box). As shown on the top (green box), CostCO generates cost models that the hybrid-protocol compiler queries during protocol selection.

be manually found and MPC protocols with cost functions that differ from ABY (e.g., [10], [20], [64]) require nontrivial effort to support. Furthermore, existing MPC protocols are actively developed on and frequently branch into different versions, e.g., ABY [15], [44], [49] and SPDZ [14], [35], [36]. Adapting existing cost models to account for new versions of protocols necessitates human effort, impeding the wider adoption and availability of MPC.

– *Limited decision-making capabilities.* In addition to being unable to support more MPC protocols, existing compilers are limited by their cost models which predict only one type of cost (runtime). As a result, they cannot make more sophisticated decisions like choosing between an MPC protocol [31] and its memory-optimized version [27].

In this work, we present CostCO, a cost modeling framework for MPC capable of automatically deriving an accurate cost model for a given MPC protocol. At a high level, CostCO takes as input an MPC protocol with a specification, builds a statistical model indicating which programs to test on the MPC protocol, measures the protocol running on these samples, and uses the results to compute a cost model. The computation of cost models in CostCO leverage the common properties of a class of MPC protocols. Specifically, security requirements of MPC imply that computation time and execution are deterministic and input-data independent, which in turn means that all branch arms are evaluated during execution (we refer to this property as non-branching). Additionally, many recent MPC protocols [3], [4], [14], [15], [18], [36], [64], [66] have quadratic communication, computation and memory complexity. In combination these insights allow CostCO to efficiently generate cost models *without* requiring additional user input such as representative workloads, etc. In contrast to prior cost modeling works [11], [33], [50], CostCO is:

– *Automatic.* A protocol provider supplies an MPC protocol implementation together with a short specification, and CostCO automatically infers a cost model, both in terms of a symbolic equation and an empirical cost for each metric. Subsequent versions of MPC protocols require less effort to generate cost models and integrate into the hybrid protocol compiler. CostCO can also automatically generate cost models for different deployment settings the secure computation takes place on, which frees the compiler from deployment assumptions (e.g., network conditions) imposed by hard-coded heuristics.
– *Extensible.* CostCO's API (§3) is rich enough to allow it to be used with a range of different MPC protocols, running in a wide variety of different deployment settings. We have successfully used CostCO to generate cost models for 7 different MPC protocols which run the gamut from ones that use arithmetic and Boolean secret sharing [15] to ones based on garbled circuits [64] and homomorphic encryption [18]. Using CostCO in each of these cases required less than 200 lines of code, demonstrating CostCO's expressivity.
– *Versatile.* In addition to generating cost models that predict an MPC protocol's runtime, CostCO also generates cost models that predict network communication and peak memory consumption, which enables compilers to additionally consider memory-optimized MPC protocols.

Improving existing compilers. We build a compiler that uses cost models generated by CostCO in order to reduce the effort that is otherwise required for manually developing cost models for an MPC protocol. At the same time, when compared to existing frameworks and their benchmarks, our compiler still produces comparable protocol assignments. By considering the different types of costs CostCO models, our compiler is also able to make more sophisticated decisions, e.g., taking into account performance degradation from memory pressure. We believe that the research on hybrid-protocol compilers can now focus on the problem of effectively searching the complex space of combinations of MPC protocols, without expending effort on cost modeling. We open-sourced CostCO [1] to bolster this research agenda.

1.1. Techniques summary

In order to generate cost models for a variety of MPC protocols, CostCO first needs a way to interface with each protocol. Such a task is challenging because different MPC protocol implementations have widely varying APIs. CostCO manages to provide a common interface in a simple API that is still rich enough to express different MPC protocols (described in §3).

Once CostCO is able to interface with a given MPC protocol, the main challenge lies in working out the relations between the primitives for the total cost. MPC protocols typically publish their asymptotic complexity, which do not contain lower-order terms. CostCO needs to first recover lower-order terms that influence the real cost to obtain the set of primitive features that influence performance. Requiring users to derive the correct set of features is both burdensome, and error prone since feature importance might itself depend on factors such as the deployment scenario. Instead, CostCO *automatically* identifies the set of important features by leveraging techniques from response-surface methodology [28] so the user only need provide the set of features that potentially influence performance (described in §4).

Next, CostCO uses the observation that the performance and resource requirements for MPC protocols are independent of input data in order to automatically generate an empirical cost model. Determining the set of important features, and generating an empirical cost model requires empirical experiments, however running an MPC circuit can be expensive in terms of runtime and resource costs. Therefore, CostCO draws techniques from the experiment design literature [8], [28], [38] and runs in two stages to reduce the overall number of empirical observations required.

Evaluation summary. We implement and port 7 MPC protocols to CostCO (§5). Each protocol takes less than 200 LOC to use CostCO (§6.1) and CostCO generates cost models with greater accuracy than prior cost modeling approaches [11], [33], [50] (§6.2). In order to demonstrate the utility of the cost models generated by CostCO, we implement a hybrid-protocol compiler along with 6 applications and find that our compiler makes comparable, if not better (up to 41% faster) protocol assignments than the current state-of-the-art hybrid-protocol compiler which uses manually derived cost models (§6.3).

2. CostCO Overview

CostCO is designed to automatically generate cost models that provide an estimate of the execution time of running a given computation using a specific MPC protocol.

2.1. CostCO Architecture

CostCO's system architecture consists of a pipeline that automatically synthesizes a cost model from a list of user-provided features for a given MPC protocol executed in a fixed deployment environment. Fig. 2 depicts CostCO's workflow. At a high level, ① CostCO receives a protocol specification for the MPC protocol and uses it to generate the exact computation experiments to run. ② CostCO then runs the experiments on the MPC protocol's implementation and ③ determines the next set of experiments to run. ④ CostCO uses the experiment results to infer the lower order asymptotic terms and finally output an empirical cost model.

CostCO is run on the deployment environment which consists of the machine or cluster at each party where

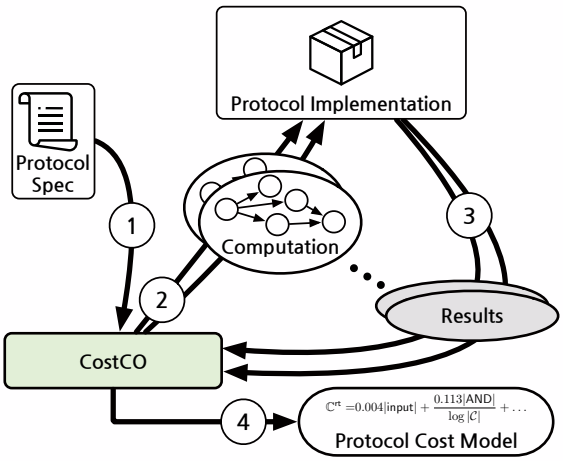

Figure 2. CostCO's workflow. CostCO is run on the deployment environment where the secure computation is planned to be executed.

the secure computation will be run and the network environment among them. For example, some banks who wish to collaborate on anti-money laundering [57] can run CostCO on their setups. For some, this means their on-premise cloud, while for others, a major cloud provider. This is the same deployment on which hybrid-protocol compilers [11], [33] are designed to run on. CostCO also computes cost models for fixed field sizes and security parameters, i.e., CostCO should be rerun if the field sizes or the security parameters change.

Similar to existing MPC literature [11], [15], [55], [64], we expect that MPC computations are expressed as *circuits*, a directed acyclic graph (DAG) where vertices represent an MPC protocol's *primitive operations*, also known as *gates* (e.g., AND/XOR in garbled circuits [64] and sum/product in arithmetic circuits [15]). The edges of the DAG represent dataflow between the gates. The *depth* of a circuit is the path length from the computation's start to end, where edges are weighted by the number of communication rounds the edge's source vertex (corresponding to a gate) requires. For example, consider a circuit for a round-based arithmetic MPC protocol [15] that consists of an addition followed by a product. In this case, the addition requires no rounds while the product requires 1 round and thus the depth of the circuit is 1.

To compute the cost model of a protocol π, CostCO requires the following inputs:

1) *Protocol specification* S_π. In order to be able to interface with π, CostCO requires a short specification for π. At a high level, a specification consists of the units of computation and parameters of π that CostCO needs in order to derive π's cost model. We describe the specification in more detail in §3.
2) *Protocol implementation*. This is a specific implementation of π, which implements the interface described in §4.2. CostCO uses this implementation to synthesize empirical cost models.

Note that, unlike prior work [11], [33], [50], CostCO only requires a list of parameters that the cost model depends upon. The list may include parameters that do not appear in the final cost model as CostCO can automatically perform parameter selection and filter out extraneous

parameters. Crucially, CostCO can derive a detailed cost model from *shallow parameters*, that is, parameters that can be easily identified without a deep understanding of the protocol. For example, in order to generate the detailed cost model for AG-MPC [64], a Boolean-circuit MPC protocol, the user inputs:

$$\left\{|\mathsf{AND}|, |\mathsf{XOR}|, |\mathsf{input}|, |\mathsf{output}|, \mathsf{depth}, |\mathcal{C}|, \frac{1}{\log(|\mathcal{C}|)}\right\},$$

where $|\mathcal{C}| = |\mathsf{AND}|+|\mathsf{XOR}|$ is the total number of gates in the circuit, and depth is circuit depth as described earlier. Of these terms $\{|\mathsf{input}|, |\mathsf{XOR}|, |\mathsf{AND}|, \mathsf{depth}, |\mathsf{output}|\}$ come from how computation in AG-MPC is represented (as a Boolean circuit) and the terms $\{|\mathcal{C}|, 1/\log(|\mathcal{C}|)\}$ appear in the asymptotic $O(\cdot)$ complexity reported in Table 1 of the original paper [64]. From the list of parameters, CostCO automatically generates the following empirical cost model for execution time in milliseconds:

$$\hat{\mathbb{C}}^{\mathsf{RT}}_{\mathsf{AGMPC}} = .004|\mathsf{input}| + .033|\mathsf{AND}| + \frac{.113|\mathsf{AND}|}{\log|\mathcal{C}|} + 56.3 \quad (1)$$

The protocol specification also only needs to be written once for an MPC protocol and can be reused for different deployment settings. The party who writes this specification can be the designer of the MPC protocol or the developer who wants to use the protocol in a hybrid-protocol compiler. We further elaborate on the shallow parameters the protocol specification writer needs to provide in §3.

After the inputs are specified, CostCO can use these inputs to generate the appropriate cost models. At a high level, CostCO's cost modeling workflow is as follows:

1) (§4.1) CostCO first carefully chooses a set of profiling experiments using experiment design. The user may optionally provide CostCO with a maximum experiment size (2^{13} operations by default) if they want to control CostCO's total execution time.
2) (§4.2) Generates computations for the profiling experiments, executes them on the implementation I_π, and collects performance measurements.
3) (§4.3) Automatically synthesizes and outputs both the *abstract* and *empirical cost models*, denoted as $\mathbb{C}_\pi$ and $\hat{\mathbb{C}}_\pi$, respectively. $\mathbb{C}_\pi$ and $\hat{\mathbb{C}}_\pi$ each contain cost models for computation, memory, and network consumption.

Compared to plaintext cost modeling systems, MPC cost modeling does not require knowledge of the input data. This is because, MPC's privacy properties mean that a program's performance *cannot* depend on data content – and generally, the overhead depends only on computation size and the input data size(s). As a result, executions are deterministic and non-branching, making cost modeling CostCO feasible with only a few additional assumptions. We elaborate on the assumptions CostCO makes and their limitations in §2.3 and §2.4.

2.2. Usage in a Hybrid-Protocol Compiler

As previously explained in §1 and Fig. 1, a hybrid-protocol compiler mixes MPC protocols $\pi_1, \ldots, \pi_n$. Parties invoke a hybrid-protocol compiler on the same deployment where they plan to run their secure computation. Since transitioning from a protocol π_i to π_j needs a special MPC conversion in order to maintain security, the compiler also needs to support *conversion protocols* $\pi_{i\to j}$, which CostCO can automatically generate cost models for.

At a high level, the hybrid-protocol compiler divides the user program into pieces and tries to determine the most cost effective way of dividing and assigning pieces of computation to MPC protocols. The exponential search space makes it impractical to directly run and measure the cost of every possible protocol assignment. Instead, the compiler infers the runtime, memory, and network requirements of each assignment by querying each of the cost models returned by CostCO.

Each empirical cost model $\hat{\mathbb{C}}_\pi = (\hat{\mathbb{C}}^{\mathsf{RT}}_\pi, \hat{\mathbb{C}}^{\mathsf{MEM}}_\pi, \hat{\mathbb{C}}^{\mathsf{NW}}_\pi)$ takes as input a *circuit file* representing a computation to be run (described in §4.2), and outputs the inferred execution time, peak memory usage, or network cost of executing the computation using π on the parties' deployment environment. The cost for executing any program can be computed by summing up the cost of executing each piece as well as the cost of conversions, which is given by plugging the data size to be converted into the cost model for the relevant conversion $\hat{\mathbb{C}}_{\pi_i\to\pi_j}$. We describe the integration of CostCO in a compiler in §5.

2.3. System Assumptions

The security properties of MPC lend themselves to appealing computational properties, namely, the execution is non-branching (all branch arms are evaluated), input-data independent, and deterministic. This means CostCO does not require additional inputs that are normally required by generic cost modelers, e.g., representative workloads to run [24], [63], [65]. CostCO also makes domain-specific assumptions that let it support automatically cost modeling many MPC protocols.

2.3.1. Quadratic approximation. As mentioned in §1, CostCO approximates a cost model as polynomial with monomials of maximum degree = 2. For many MPC protocols' cost models [3], [4], [14]–[16], [18], [36], [46], [58], [66], a quadratic approximation is sufficient. This is due to the computation time and the number of communication rounds of an MPC circuit normally being bounded by the size of the circuit ($|\mathcal{C}|$), and in the worst case no gate requires more than a constant number of all-to-all communication rounds, bounding the number of messages and hence bandwidth requirements to $|\mathcal{C}|^2$. While CostCO can approximate non-polynomial cost terms as second degree polynomials, for some protocols such as AG-MPC [64] explicitly specifying the asymptotic terms can improve accuracy, especially when they contain non-polynomial factors (e.g., $1/\log|\mathcal{C}|$). As a result, for such protocols, CostCO allows users to provide non-polynomial terms appearing in a protocol's asymptotic complexity as input and uses this information to improve cost model accuracy by including them in the set of features considered when synthesizing cost models.

This assumption also implies that the cost model is smooth (differentiable everywhere). That is to say, the protocol's empirical cost model is not piecewise; the constant factors are expected to not arbitrarily change with respect to computation size. However, CostCO can still enable the compiler to make decisions on piecewise costs.

For example, CostCO can extrapolate the performance degradation from running out of memory by artificially limiting the system's memory and measuring the difference in runtime of a circuit. By using the memory model generated by CostCO in tandem with the performance under memory pressure, our compiler can account for a protocol assignment's performance degradation when its peak memory usage exceeds the available system memory (§5).

2.3.2. Fixed parameters. CostCO runs on the deployment that the users plan to run their secure computation on, which is the same deployment existing hybrid compilers [11], [33] run on. Cost models are created for fixed field sizes and security parameters, which allows the cost models to be viewed as a function of computation size and input data size. CostCO should be rerun if the field sizes or security parameters change.

2.4. Limitations

2.4.1. Circuit structure. CostCO only considers the number of gates in the circuit and its depth. For the MPC protocols we consider, circuit depth corresponds to the number of communication rounds required when evaluating the computation. We show in §6 that considering these factors is sufficient for producing relatively accurate cost models, despite not considering round-level parallelism in protocols with non-constant communication rounds.

2.4.2. Circuit optimization. The cost models generated by CostCO are derived from mapping several unmodified circuits (§4.2) to the costs experienced during their execution. In practice, some protocols [36] can reorder the evaluation order of the units comprising the circuit to improve performance. Significant modifications could render analysis on the unmodified circuit useless to extrapolate to the modified circuit which the protocol actually evaluates. In order to produce accurate models for these protocols, CostCO would need a way to execute a circuit without optimizations from the protocol. The protocol would need to implement an interface that takes the computation and outputs the optimized circuit that can be used as input to the cost model.

2.4.3. Scheduling. MPC protocols may use multiple threads throughout their execution. The runtime of these threads is at the mercy of the scheduler while CostCO is profiling costs. Scheduling decisions may influence the overall runtime performance of the protocol, but CostCO makes no attempt to understand such scheduling decisions.

3. CostCO Specification

The first challenge that CostCO needs to address is how to interface with a given MPC framework. MPC frameworks differ significantly from each other both in how computation is expressed and in how it is executed. CostCO manages to provide a common interface using a simple API. We observe that many MPC protocols express computation as a sequence of *primitive operations*, also known as *gates*. A primitive operation is either a *computational gate* or a *data gate*. For example, the primitive operations of AG-MPC are the AND and XOR gates (computational units), and the input and output gates (data units). CostCO assumes the cost model can be computed as a function of the MPC protocol's computational primitive operations. Intuitively, because MPC requires the execution to be non branching, deterministic, and input-data independent, the cost of an MPC protocol can be viewed in terms of the computation itself. The specification for an MPC protocol π captures the primitive operations in π.

Given the specification, CostCO will automatically generate experiments to evaluate the MPC framework, collect the results, and derive the abstract and the empirical cost models.

A protocol specification S_π for protocol π is a file that details the following:

1) A set of *gates*, each of which is a computational gate or data gate and has the form $\mathsf{g_n} = (\mathsf{i_n}, \mathsf{o_n}, d_n)$, where n is the name of the gate, $\mathsf{i_n}$ is the number of inputs, $\mathsf{o_n}$ is the number of outputs, and d_n is the depth. Depth here indicates the number of communication rounds required to run this computational unit. We define $\mathcal{G}$ to be the entire set of π's gates. Inputs and outputs are assumed to be gates, i.e., $\mathcal{G}_{\mathsf{io}} = \mathsf{input}, \mathsf{output}$ and $\mathcal{G}_{\mathsf{io}} \subset \mathcal{G}$. For a given computation, we define $\mathcal{S} = \{s_{\mathsf{g}}\}$ to be the counts of every gate $\mathsf{g} \in \mathcal{G}$ in that computation.
2) A set of *asymptotic terms*, each of which is some function $h : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}$ that expresses parameters in the cost model as a function of some subset of $\mathcal{S}$ and appears as a term in the worst-case asymptotic complexity bound $O(\cdot)$ of π. We define $\mathcal{H}$ to be the entire set of π's asymptotic terms. The counts of gates $\mathcal{S}$ and the asymptotic terms $\mathcal{H}$ evaluated on $\mathcal{S}$ make up the set of features $\mathcal{F} = \{f_1, \ldots, f_{|\mathcal{F}|}\} = \mathcal{S} \cup \{h(\mathcal{S}) | h \in \mathcal{H}\}$ that CostCO uses to derive a cost model. The cost model is expected to have the form:

$$\mathbb{C}_\pi(f_1, \ldots, f_{|\mathcal{F}|}) = \mathbf{r}^\top \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ f_{|\mathcal{F}|} \end{bmatrix} + \mathbf{s}^\top \begin{bmatrix} f_0 f_0 \\ f_0 f_1 \\ \vdots \\ f_{|\mathcal{F}|} f_{|\mathcal{F}|} \end{bmatrix},$$

where $\mathbf{r}_i, \mathbf{s}_i \in \mathbb{R}^+$.

Example. To make the specification writing process more concrete, we will elaborate on the specification for AG-MPC [64] introduced in §2.1. Boolean MPC requires the computation to be specified as a Boolean circuit composed of XOR and AND operations. Note that since CostCO assumes that the deployment is fixed, variables such as the number of parties and network bandwidth are also fixed and thus integrated into the empirical constants of the cost model. Both operations have two inputs and one output. AG-MPC is a constant-round protocol, thus both operations have a depth of 0. Therefore, AG-MPC has the following gates:

$$\mathcal{G} = \{(\mathsf{XOR}, 2, 1, 0), (\mathsf{AND}, 2, 1, 0)\} \cup \mathcal{G}_{\mathsf{io}}.$$

Table 1 of AG-MPC's original publication [64] lists an asymptotic complexity of $O(|\mathcal{C}|/\log|\mathcal{C}|)$. Because

Algorithm 1: Cost model synthesis.

Input: $\mathcal{F}, \mathcal{H}, d$
Output: a cost model $(\mathcal{F}^*, \beta^*)$ for π

```
1  begin
2      F̃ ⟵ ∅
3      X1 ⟵ PBD(|F|)
4      Y1 ⟵ RunCircuitFiles(X1)
5      β ⟵ LeastSquares(X1, Y1)
6      for f_i ∈ F do
7          if 0 ∉ ConfInterval(β_i) then
8              F̃ = F̃ ∪ {f_i}
9          end
10     end
11     X2 ⟵ CCD(|F̃|)
12     Y2 ⟵ RunCircuitFiles(X2)
13     G = F̃ ∪ H, G^2 = {g_i · g_j | g_i, g_j ∈ G}
14     F*, β* ⟵ FoBa(X2, Y2, G ∪ G^2)
15 end
```

CostCO is capable of automatically inferring quadratic parameters (monomial degree = 2),

$$\mathcal{H} = \left\{ |\mathcal{C}|, \frac{1}{\log |\mathcal{C}|} \right\}.$$

4. Synthesizing Cost Models

Given an MPC protocol specification and implementation (§3), the goal of CostCO is to compute a cost model for that protocol. When designing CostCO's cost modeling algorithm (Algorithm 1), we initially considered using techniques from optimal experiment design (OED) [33], which provides a methodology for choosing samples while minimizing some desired metric (e.g., metrics related to expected error of the estimator). Existing cost modelers like Ernest [63] leverage OED to predict the performance of large-scale data analytics workloads. However, OED requires a known model which, in the case of AG-MPC above, would require knowing the abstract cost model *a priori*:

$$\mathbb{C}^{\mathsf{rt}}_{\mathsf{AGMPC}} = x_0 + x_1|\mathsf{input}| + x_2|\mathsf{AND}| + x_3\frac{|\mathsf{AND}|}{\log|\mathcal{C}|} \quad (2)$$

Generating such a model is challenging for a user because it requires a non-trivial understanding of the protocol and in some cases might rely on implicit assumptions about the deployment (e.g., network latency). While it is hard for the user to specify the exact abstract cost model equation, it is easier for the user to come up with factors that affect performance. For example, as previously mentioned in §2.1, MPC protocols commonly report their asymptotic $O(\cdot)$ complexity. Thus, CostCO asks the user to only specify such factors (the user can also provide more than is necessary) and does not require the user to figure out how to combine the factors. CostCO blends techniques from experiment design and statistics to cull important features (§4.1), generate and run a set of experiments (§4.2), and perform regression analysis to construct the abstract and empirical cost models (§4.3).

	Features		
Experiment	f_1	f_2	f_3
1	1	1	1
2	1	0	0
3	0	1	0
4	0	0	1

Figure 3. PBD for three features and two levels. After running every experiment in PBD, CostCO can analyze which features f_i to drop before the second phase of more intensive experiments.

4.1. Design of Experiments

Given a set of potential features, CostCO still needs to figure out which experiments to run. As mentioned in §4, we cannot use OED because we do not know the abstract cost model (e.g., Equation 2) *a priori*. Other techniques that predict execution time using random sampling [29], [60] do not provide a principled rule on the amount of samples to collect and ended up needing hundreds of samples to train a model. CostCO's approach is inspired by ideas from response-surface methodology (RSM) [28], a technique used in several disciplines to explain how different features influence an observed value. We remain sample efficient by leveraging two techniques in the experiment design literature used in tandem with RSM.

CostCO initially starts with a set of user-provided potential features $\mathcal{F}$. In our running AG-MPC example,

$$\mathcal{F} = \{|\mathsf{AND}|, |\mathsf{XOR}|, |\mathsf{input}|, |\mathsf{output}|, \mathsf{depth}, |\mathcal{C}|, 1/\log(|\mathcal{C}|)\}.$$

In this subsection, we will be working on $\mathcal{F}_M$, the monomial subset of $\mathcal{F}$. That is,

$$\mathcal{F}_M = \{|\mathsf{AND}|, |\mathsf{XOR}|, |\mathsf{input}|, |\mathsf{output}|, \mathsf{depth}\}.$$

In order to identify the important features $\tilde{\mathcal{F}}$, CostCO carries out a screening step (Lines 3–10), that generates a set of experiments used to compute the importance of each potential feature. A naïve way to find $\tilde{\mathcal{F}}$ is via a full factorial experiment [22], which is a design of size $2^{|\mathcal{F}_M|}$ that enumerates all permutations where each feature is either at its highest value or lowest value. Instead, CostCO uses Placket-Burman design (PBD) [51] for this step because it produces a set of experiments that is small but still capable of finding features that have a significant effect on the runtime. PBD is a fractional factorial design requiring only $4\lceil (n+1)/4 \rceil \approx n+1$ measurements for n features, under the assumption that interactions between features (second-order) are confounded with main (first-order) effects. A PBD for $n = 3$ is shown in Fig. 3. The samples collected from PBD are fit and features are filtered according to the confidence intervals computed with Equation 3. After the screening step in our running example,

$$\tilde{\mathcal{F}} = \{|\mathsf{AND}|, |\mathsf{input}|\}.$$

The screening step using PBD indicates that depth does not have a statistically significant impact on performance, and we drop that from the list of features. This is because AG-MPC is a constant-round protocol, and circuit depth does not impact runtime. |XOR| is dropped because XOR requires no additional communication to compute and

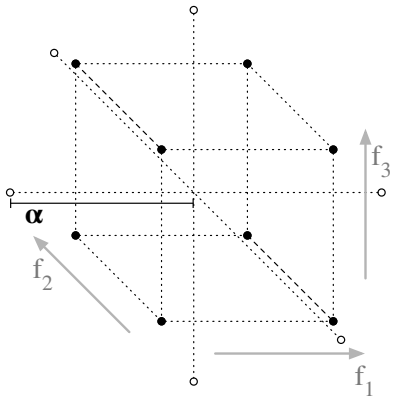

Figure 4. CCD for three features. Star points (unshaded points) are α away from the center of the factorial design (shaded points). Note that α can be a different maximum value for each feature.

|output| is dropped because its effect is both small and limited by the circuit size.

CostCO then carries out a more intensive experiment step using the culled set of features (Lines 11-12). CostCO uses central composite design (CCD) [8], an experiment design methodology that is more expensive than PBD but capable of capturing interactions between features in $\tilde{\mathcal{F}}$ (second-order effects). A CCD is composed of a factorial design augmented with star points that are a distance α from the center [8]. Fig. 4 shows a CCD for $|\tilde{\mathcal{F}}| = 3$. The extra sampling points help build a second-order model for the features without having to carry out a complete three-level factorial experiment ($3^{|\tilde{\mathcal{F}}|}$ samples). CostCO is also able to reuse data from the PBD experiments (e.g., PBD forms a subset of the cube's vertices in Fig. 4) instead of re-running those experiments. CostCO then runs these experiments (§4.2) to obtain data for computing the abstract and empirical cost models (§4.3).

4.2. Generating and Running Experiments

In the previous section, we explained CostCO's two-phase experiment design procedure. Given the set of features and their values each experiment, CostCO still needs to use the user-provided protocol specification (described in §3) and the implementation to run one or more profiling experiment instances. In order to do so, given a profiling experiment CostCO synthesizes and runs computational circuits with the required features (Lines 3-4 and 11-12).

4.2.1. Circuits. Similar to many MPC frameworks, CostCO represents computation as a *circuit*. Each circuit is a DAG, where the vertices, which we call gates, correspond to the π primitive operations; and edges represent data flow between gates. An edge $g_i \rightarrow g_j$ means that the output from gate g_i is used as an input to g_j. In addition, CostCO needs to ensure that generated circuits can capture properties like network round-trips, etc. Protocols that are not constant-round, e.g., arithmetic and Boolean ABY [15], have performance that can depend on the sequence of operations that are run. CostCO models this dependency in the circuit as the depth d_n of a primitive operation; this information is supplied as a part of the protocol specification. If g_i represents a gate with depth d_i, then all outgoing edges from g_i are assigned the weight d_i. The depth of the entire circuit is the number of round trips needed to run the computation, which is the equivalent to the length of the longest path in the circuit.

4.2.2. Generating circuits. CostCO generates representative computation in the form of a circuit. The circuit generation component needs to take as input the number of each gate (computational gates and input gates) and the desired circuit depth. Given these parameters, CostCO constructs the circuit by first putting in input gate according to the input size. While building the circuit, CostCO keeps track of: (1) gates that still need to be realized on the circuit and (2) gates with available outputs that can be used as an input to another gate. CostCO selects a gate from the first list and places it on the circuit. Using the second list, inputs are selected for the new gate. CostCO repeats this process until all gates have been realized in the circuit. CostCO achieves the desired circuit depth by keeping track of path lengths and preventing edges that would result in a path length exceeding the desired depth. Finally, CostCO ensures that the output of every gate is either fed into another gate as input or is evaluated as an output.

Note that the input to CostCO's circuit generation component (number of gates and desired circuit depth) are not specified by the user but are automatically generated and varied by CostCO's experiment design component (§4.1).

4.2.3. Running circuits. CostCO assumes that the MPC framework can accept inputs in the circuit format described above. CostCO therefore expects that the protocol provider implements the following function:

> `RunCircuit(CircuitFile cf)` Execute the circuit represented by `cf` with the protocol π and report the total execution time and network communication.

CostCO generates circuits and calls the protocol's implementation of `RunCircuit` to profile its performance with circuits generated from the methodology described above. Empirically, implementing `RunCircuit` took less than 200 LOC for every MPC protocol we evaluated.

4.2.4. Types of cost measured. CostCO generates cost models for three types of execution costs: runtime, peak memory usage, and network communication. CostCO opts to measure memory in addition to runtime and network communication because π can experience a significant performance degradation if the peak memory of an execution exceeds available memory. Recent protocols have started trading off more communication rounds for using less memory [70].

4.3. Deriving Cost Models

After choosing and running profiling experiments, CostCO selects terms from the monomial culled features $\tilde{\mathcal{F}}$, asymptotic terms $\mathcal{H}$, and their pairwise products in order to form the final abstract cost model (Lines 13–14). In our running example,

$$\mathcal{G} = \tilde{\mathcal{F}} \cup \mathcal{H} = \{|\mathsf{AND}|, |\mathsf{input}|, |\mathcal{C}|, 1/\log(|\mathcal{C}|)\}$$

$$\mathcal{G}^2 = \left\{ |\mathsf{AND}|^2, |\mathsf{AND}||\mathsf{input}|, |\mathsf{AND}||\mathcal{C}|, \frac{|\mathsf{AND}|}{\log(|\mathcal{C}|)}, \ldots \right\}.$$

We make the observation that only a handful of the terms in $\mathcal{G}$ and $\mathcal{G}^2$ contribute significantly to the cost. Intuitively, this is because many of the pairwise products do not appear in the asymptotic $O(\cdot)$ cost. Thus, the problem becomes one of selecting a *sparse* polynomial out of $\mathcal{G} \cup \mathcal{G}^2$. CostCO uses ordinary least squares regression and leverages the adaptive forward-backward (FoBa) greedy algorithm described in [68] to select a sparse polynomial. FoBa first greedily adds terms to the candidate polynomial in the forward step, halting when the reduction in mean error of adding a feature is less than ϵ. To correct for incorrect features added in the forward step, FoBa's backward step removes a feature if the increase in mean error is less than a factor δ than the last decrease in mean error.

To test how well the polynomial produced by FoBa generalizes, CostCO runs cross-validation on the outputted model. Specifically, CostCO uses leave-one-out-cross-validation where one data point is left out and the leftover data is re-fit to the polynomial. This is repeated for all points in the data. Since FoBa is parameterized by the stopping threshold ϵ, CostCO uses the mean error from the cross-validation runs as a metric when binary searching for the optimal ϵ.

In order to further prevent selecting a polynomial that overfits the data, we only consider a model (polynomial outputted by FoBa) if all of its features have regression coefficients with a positive 95% confidence interval. Under the least squares assumption that observation errors are normally distributed, i.e., $\varepsilon \sim \mathcal{N}(0, \sigma^2)$, the estimated coefficients $\hat{\beta}_i$ from samples X have a $100(1-\alpha)\%$ confidence interval of:

$$\hat{\beta}_i \pm t_{\alpha/2, n-(d+1)} \sqrt{\hat{\sigma}^2 [(X^\top X)^{-1}]_{i,i}}, \tag{3}$$

where $\hat{\sigma}^2$ is the mean squared error (estimate of σ^2), t is the Student's t-distribution, and $n-(d+1)$ is the degree of freedom. Note that we make the least squares assumption because MPC protocols are usually deterministic in their execution—dependence on data content would otherwise cause privacy leakage—so errors stem from the execution environment.

5. Implementation

We implemented a prototype of CostCO and a hybrid-protocol compiler that uses the generated cost models in ≈5000 lines of code (LOC) of Python. Porting MPC protocols to CostCO took less than 200 LOC per protocol (§6.1). Lines of code were counted with `cloc`. Similar to the SPDZ [36] compiler, our compiler takes programs written in a subset of Python. It currently supports six MPC protocols: Arithmetic, Boolean, Yao (and their share conversion protocols) [15]; FastGC [31]; AG-MPC [64]; and SPDZ [14].

We use a program decomposition approach and a protocol conversion assignment approach based on OPA [33]. However, instead of treating the optimal protocol assignment problem as a linear program, we use a randomized greedy algorithm that reduces the search space by avoiding conversions with some probability p if the cost of converting shares to a different protocol outweighs the immediate cost reduction from using a different protocol. This allows us to mix all three protocols (Arithmetic, Boolean, and Yao) whereas the linear program relaxation, though integral, only allows two protocols to be mixed at a time [33]. We leave finding better search algorithms for the optimal program assignment problem for future work.

The cost models provided by CostCO also enable our compiler to consider peak memory consumption when assigning protocols for a program and select between variants of an MPC protocol like FastGC [31], which is a memory-optimized version of Yao [15].

6. Evaluation

We evaluate CostCO on 7 MPC protocols. We first report on the ease of porting MPC protocols to CostCO (§6.1). We then evaluate the accuracy of our generated models (§6.2.1) and how they compare to current work in cost modeling (§6.2.2). We conclude with results on the cost of running CostCO (§6.2.3).

We ran all evaluations on AWS `c5.4xlarge` instances (16 vCPUs and 32GB RAM) in the same region. All protocols used a deployment with 2 parties except AG-MPC, which used 3 parties. All protocols, when applicable, had their field sizes set to 32 bits. We set the maximum number of a feature per experiment to be the default value of 2^{13}. This value determines the concrete value to set for a feature at the "high" point in a factorial design (described in §4.1).

Additionally, we implemented our own hybrid-protocol compiler for ABY based off the techniques in [33]. We replace the cost models used by [33] with the cost models automatically generated by CostCO and find that it arrives at the same hybrid protocol assignments for the applications we tested (GCD, biometric matching, and non-parallelized k-means). In all cases we found that the automatic models generated by CostCO allowed the compiler to perform as well as a compiler that uses hand-tuned, human generated cost models.

6.1. Ease of Use

We integrated the following MPC protocols to work with CostCO: the three in ABY [15] (Arithmetic (A), Boolean (B), and Yao (Y)), their conversions (A→Y, B→Y, B→A, Y→A, A→B, Y→B), FastGC [27], AG-MPC [64], SPDZ [36], and BFV homomorphic encryption [18]. Fig. 5 shows the lines of code needed to implement the required functionality for the protocol using CostCO (§3), as well as the (existing) lines of code in the protocol's implementation.

Running circuits. In order to interface with CostCO, the MPC protocol needs to implement the `RunCircuit` API described in §4.2. Implementing `RunCircuit` took about 1-2% of a protocol's implementation size.

Protocol spec. The gates for these protocols were simple to enumerate (Boolean vs. arithmetic operations), resulting in a compact specification of 20 lines or less. The degree bounds on runtime and communication complexity were

Protocol	Existing	Using CostCO	
	Implementation	`RunCircuit`	Spec
ABY [15]	13722	161	62
A	—	141	14
B	—	4	14
Y	—	4	14
A→Y, B→Y, B→A, Y→A, A→B, Y→B	—	12	20
FastGC [31]	9905	139	14
AG-MPC [64]	7269	100	14
SPDZ [14]	39511	185	166
BFV [58]	8854	164	14

Figure 5. Lines of code to generate cost models using CostCO for each MPC protocol, split between implementing `RunCircuit` and writing the MPC protocol specification. Using CostCO requires additional code that is 1-2% of the existing MPC protocol's implementation size.

straightforward to compute from the asymptotic complexity results published for each protocol. The only protocol with a non-polynomial asymptotic term in its cost model was AG-MPC, and its term $|\mathcal{C}|/\log|\mathcal{C}|$, which appears in its reported asymptotic complexity (see Table 1 of [64]), can be input into CostCO in a straightforward manner ($|\mathcal{C}|$ and $1/\log|\mathcal{C}|$). In contrast, we found that manually deriving the cost model to be demanding and error-prone due to requiring further analysis of the protocol. While the term $|\mathcal{C}|/\log|\mathcal{C}|$ is readily available in the reported asymptotic complexity, users cannot uncover additional terms in its cost model (such as $|\mathsf{AND}|/\log|\mathcal{C}|$) without more detailed analysis of the protocol.

6.2. Microbenchmarks

6.2.1. Model Accuracy. To evaluate how well the models produced by CostCO extrapolate, we generate cost models for runtime, network usage, and peak memory usage; and determine their accuracy in predicting each metric on a set of 100 circuits with features of size up to $8\times$ larger (2^{16}) than the largest feature size in the samples used to build the cost model. We sample the test circuits randomly with a fixed seed.

The Root Mean Squared Error (RMSE) as well as the 5th, 50th, and 95th percentile of the relative error of the runtime cost models produced by CostCO for the 3 MPC protocols in ABY [15] are reported in Fig. 6. The relative error as a percent value is calculated as:

$$\left|\frac{\mathsf{actual}-\mathsf{predicted}}{\mathsf{actual}}\right|\times 100.$$

6.2.2. Comparison to Existing Cost Models. CheapSMC [50]. CheapSMC computes cost models by constructing a separate circuit with 1000 operations for every operation in ABY. CheapSMC then records the runtime of each circuit and divides the average over 10 runs by 1000 to get the average per-operation cost. The cost of running a circuit becomes the sum of its individual operation costs. This approach, while efficient and easy to reason about, sacrifices model accuracy because the structure of the circuit is ignored. This can be seen in Fig. 6, where the median accuracy of Yao is $28\,\%$ because while the circuits are evaluated in topological order, the depth of the circuit does not affect the number of communication rounds. In Arithmetic and Boolean, however, the median relative error degrades to $388\,\%$ due to the circuit depth affecting the number of communication rounds.

Optimal Mixing (OPA) [33] & HyCC [11]. The cost model used by OPA builds on the methodology of CheapSMC by also considering circuit structure. This is accomplished by running circuits of each gate with different levels of parallelism ($n \in \{1, 2, 5, 10, 25, 50, 100, 200, 300, 500, 800\}$). The cost of a circuit is calculated by finding the number of gates running at each level and taking the appropriate per-gate cost at that level of parallelism. This method suffers from compounding errors introduced from the overhead of executing an actual circuit, e.g., to run a circuit with one gate, the circuit also needs two inputs and one output. HyCC uses a similar cost modeling approach, except with only 4 levels of parallelism.

Note that *relative* accuracy matters for protocol selection, which we further investigate in §6.3.

6.2.3. Cost of Model Generation. We now characterize the economic cost of generating our cost models. Our average runtime to generate the cost models (displayed in Fig. 7) for our set of circuits for each of our profiled protocols was in general less than two and a half minutes with an approximate worst-case addition of one minute to profile the memory usage. This leads to a total cost (in our `r5.xlarge` deployment) of model generation of less than $\$0.005$ for each protocol. Although other hybrid-protocol compilers are comparatively cheaper, CostCO is able to automatically generate the experiments to run, saving on manual effort.

6.3. Application Benchmarks

6.3.1. Comparison to Related Work. In order to demonstrate the utility of the cost models automatically generated by CostCO, we compared our compiler to prior benchmarks from HyCC and ABY. Below, we describe and discuss each application. The assignments made by each compiler are summarized in Fig. 8 and the runtimes of each application are illustrated in Fig. 9.

Modular exponentiation. Two parties want to compute $x^y \mod m$, where one party holds x and the other holds y (all values are 32-bit integers). CostCO computes an assignment similar to the manual ABY [49] assignment.

Biometric matching. One party holds a list L containing m n-dimensional vectors and the other holds an n-dimensional query vector q. The two parties come together to compute the vector $l \in L$ with the shortest Euclidean distance from q. For $(m = 256, n = 4)$, CostCO computes an assignment that is in total $\approx 40\,\mathrm{ms}$ slower than HyCC. This difference is due to HyCC's implementation of biometric matching, which structures the computation

	CostCO				CheapSMC [50]				Opt. Mix [33]			
Protocol	p5	p50	p95	RMSE	p5	p50	p95	RMSE	p5	p50	p95	RMSE
A	1.01	15.0	31.1	37.5	224.7	1163.2	1559.6	2952.4	84.3	238.4	408.8	549.8
B	0.6	10.6	33.5	150.6	19.0	387.8	566.7	4771.5	71.1	314.7	612.8	3632.1
Y	1.2	5.3	13.5	172.9	2.1	28.0	62.9	394.4	313.4	495.2	719.7	5732.3

Figure 6. Comparison of each cost modeling framework's runtime prediction error (%) and $\sqrt{MSE}$ (ms) on ABY [15]. The percentiles are computed from prediction errors of a set of 100 randomly generated circuits.

	CostCO			CheapSMC [50]		Opt. Mix [33]		HyCC [11]	
		Runtime	Memory		Runtime		Runtime		Runtime
Protocol	(#)	(sec)	(sec)	(#)	(sec)	(#)	(sec)	(#)	(sec)
A	34	143.1	62.9	2	14.7	20	35.1	1	0.16
B	34	120.2	69.1	2	8.8	20	80.0	1	0.16
Y	27	111.4	51.0	2	7.6	20	82.1	1	0.16

Figure 7. Number of experiments (#) and total time (sec) taken by each cost modeling framework to collect samples.

Benchmark (LAN)	**Best Known**	**CostCO**	**HyCC***	**HyCC†**	**ABY (Manual)**
Modular exponentiation	A+B+Y	A+B+Y	—	—	A+B+Y
Biometric matching	A+Y	A+Y	A+B	A+Y	A+Y
Private set intersection	Y	Y	B	Y	—
k-means	A+Y	A+Y	A+B	A+Y	—
DB-Merge	A+Y	A+Y	A+B	A+Y	—
MiniONN MNIST	A+Y	A+B+Y	*	A+Y	—

Figure 8. Benchmark comparisons with HyCC [11] (automatically generated) and ABY [15] (manually written). HyCC* uses their cost model and protocol selection algorithm, which did not finish running after 2 weeks for MiniONN. HyCC† uses their heuristic approach, which selects the best out of 5 default protocol assignments. The "Best Known" assignment is determined by measuring the runtimes of each assignment and taking the assignment with the minimum runtime.

of the minimum as a tree, minimizing the depth of the computation.

Private set intersection. One party holds a set S_1 and the other holds a set S_2. The two parties want to compute the intersection of S_1 and S_2 using the standard $O(n^2)$ algorithm. CostCO outperforms the automatic protocol selection of HyCC by 84 ms but is 595 ms slower than the hand-selected HyCC protocol. This is due to HyCC running a circuit-level minimization tool which CostCO does not run, which results in a circuit that is 41% smaller.

Database merge. Both parties hold a database with two columns and want to compute the aggregate mean and variance of their merged databases. CostCO outperforms both the automatic HyCC assignment (by 1.28 s) and hand-picked HyCC assignment (by 560 ms). This is due to CostCO also assigning the arithmetic MPC protocol when computing the squares of differences needed to calculate the variance. HyCC only assigns the arithmetic MPC protocol for addition operations when computing the mean and variance.

k-means. Both parties hold datapoints and want to identify centroids in their data using a textbook k-means algorithm [42]. CostCO outperforms the automatic HyCC assignment (by 3.1 s) and hand-picked HyCC assignment (by 2.5 s). Note that the implementation of k-means in CostCO is more straightforward than HyCC's implementation, which decomposes into multiple inner and outer loops.

Secure prediction. The last application is a machine learning (secure prediction) workload where one party holds the model and the other party holds an input for the model. The two parties want to compute the input's corresponding prediction from the model. We implemented the convolutional neural network described in MiniONN [41] which performs a secure prediction with a neural network trained on the MNIST dataset. HyCC's performance is affected by a bug where certain repetitive calls to a function result in empty circuits (i.e., less work and lower runtime compared to CostCO's debugged assignment). We accounted for this by removing the computations that were removed by HyCC. In this context, HyCC's solver did not finish running and CostCO was able to produce an assignment that was 41% faster than HyCC's hand-selected assignment. The rightmost bar in Fig. 9 shows the runtime of debugged MiniONN (it does not lose any work), which has a runtime 40% higher than HyCC's buggy MiniONN.

6.3.2. Memory. The cost models produced by CostCO enable our compiler to make more sophisticated decisions by considering each MPC protocol's peak memory usage. To emulate a resource-constrained device, we set

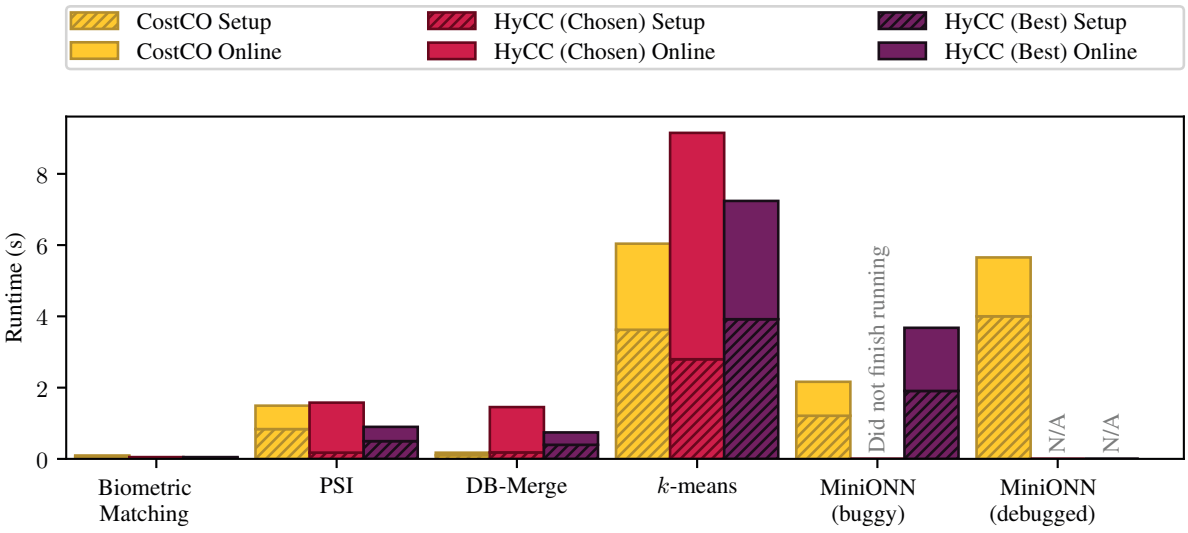

Figure 9. Runtime breakdown and comparison of hybrid protocols generated by CostCO and HyCC [11] for various applications. Each bar is broken up into Setup (shaded) and Online runtimes.

MPC Protocol	CostCO	HyCC [11]	OPA [33]
ABY [15]	✓	✓	✓
FastGC [31]	✓	✗	✗
FastGC-Mem [27]	✓	✗	✗
AG-MPC [64]	✓	✗	✗
SPDZ 2.0 [36]	✓	✗	✗

Figure 10. MPC protocol support across hybrid-protocol compilers.

an artificial memory budget of 200MB using `cgroupv2` and informed the compiler of the memory limit. This caused CostCO to choose the memory-optimized version of FastGC for our DB merge application. Note that other compilers like HyCC do not model memory consumption, and hence cannot make the correct decision in choosing between an MPC protocol implementation [31] and its memory-optimized version [27].

6.3.3. Extensibility. The most recent hybrid-protocol compilers only support the 3 protocols in ABY [49], which provides MPC for 2 parties and semi-honest security (no party deviates from the protocol). Our compiler is able to use CostCO's cost models to incorporate additional MPC frameworks (SPDZ, AG-MPC, FastGC), which lets it choose MPC protocols for a variety of usage settings, e.g., more than 2 parties and malicious security. Fig. 10 lists the MPC protocol support across hybrid-protocol compilers.

7. Related Work

7.1. MPC compilers

MPC compilers originated as a means to enhance the accessibility of secure computation outside of expert users. Developers often lack the domain-specific knowledge required to employ MPC. These compilers allow developers to describe secure computation through high-level languages, obfuscating the underlying details of the protocols. Fairplay [5] represented the first attempt at compilation, proposing a domain specific language that compiles to a garbled circuit. This work launched the compiler space leading to a wealth of future work. Subsequent single protocol compilers have focused on optimizing performance or scalability (e.g. Sepia [9], Sharemind [6], TinyGarble[61] FastGC [2]), and improvements upon the high-level abstraction (e.g. Picco [69], CMBC-GC [19], Obliv-c [67], ObliVM [40], Wysteria [53]).

As MPC protocols began to specialize, performing especially well for a single type of computation, the need arose for hybrid compilers that leverage each of these categories of protocol. These hybrid protocol compilers have the additional task of partitioning the program and choosing the optimal MPC protocol for each piece of the computation. As a result, each compiler either relies on hard-coded heuristics or on cost models to guide their decision. Tasty [26] was the first such hybrid protocol compiler and allowed the generation of secure protocols combining homomorphic encryption and garbled circuits. In this framework, the developer hard-codes the protocol to be used for each operation. After Tasty, there have been a number of different approaches to protocol selection. Some frameworks, such as EZPC [13] include hard-coded heuristics for their currently supported protocols and require coming up with new heuristics when integrating new protocols. Other recent works profile the cost of the gates and model the total circuit cost as a sum of the gates (HyCC [11], CheapSMC [50], OPA [33]).

7.2. Cost Modeling

There have been a number of works on cost modeling outside of secure computation that still hold parallels to the MPC environment. Recently, with the increasing distributed nature of computation, many works have tried to quantify the performance of jobs as a function of a cloud

configuration while minimizing the individual experiments required. Ernest [63] aims to enable efficient performance prediction based on the assumption that the related jobs have a predictable structure. The system leverages optimal experiment design [52] to minimize the number of samples required to develop such a predictor. Generic empirical cost modeling frameworks like `trend-prof` [24] receive a representative workload as input from the user and fit to a pre-determined type of model (linear or powerlaw). CostCO leverages the security properties of MPC to produce cost models that are independent of data content.

7.3. Statistics

Experiment design [38] is a widely applicable statistical tool to determine the features responsible for a value. It allows the user to maximize the understanding gained per-experiment of the relationship between the response variable and different features. Optimal experiment design [52] allows a value to be estimated with minimum variance and bias. In a multi-parameter setting the variance of the parameter estimator is a matrix. Its inverse is denoted as the information matrix. There are different categorizations of optimality based upon minimizing different values related to the information matrix.

There are several instantiations of optimal experiment design. One such category of implementations is iterative experimentation which concerns the development of sequential experiments. Response-surface methodology [28] is an iterative experimentation used to optimize the response by exploring the surface of the response curve. First, in order to explore the curve when far from the optimum, the experimenter uses the method of steepest ascent on a first order model to find the most efficient direction to move. When closer to the optimum, the experimenter switches to a second order model that is more expensive to compute (because more factors equates to more runs), but more accurate. In order to minimize the number of runs required to fit the second-order model, RSM employs the uses of factorial designs such as CCD [8]. A factorial design is a design that enumerates all permutations of feature settings where each feature is either at its highest value or lowest value. An example of a fractional factorial design is Plackett-Burman design (PBD) [51], which has been shown to be particularly useful in unconstrained configuration spaces [60].

Sparse representations in functional relationships prevent overfitting to the dataset. Traditionally, sparse learning has been performed by using one of three approaches: lasso regularization; forward greedy algorithms, which choose a feature to add to maximize the reduction in the cost function; and backward greedy algorithms, which choose features to minimize the increase in the cost function. FoBa [68] employs the uses of both greedy algorithms showing reduced training error to the three classic choices making sure backward steps don't erase the gain made in forward steps. They ensure that backward steps are only taken when the cost functions increase is less than or equal to half of the decrease of the cost function in earlier forward steps.

8. Conclusion

The recent growth in the development of MPC protocols has significantly improved the performance and feasibility of MPC. However, it has led to a zoo of MPC protocols that a prospective user must reason about when choosing a protocol for their workload. CostCO helps compute accurate cost models for different protocols and does so in an automated way. Accurate cost models can aid in the synthesis of efficient hybrid MPC protocols, which could enable the realization of significantly increased employment of practical secure computation.

References

[1] CostCO repository. https://github.com/ucbrise/costco.

[2] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[3] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Proceedings of the Annual International Cryptology Conference (CRYPTO)*, 1991.

[4] Donald Beaver, Shafi Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1990.

[5] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: A system for secure multi-party computation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[6] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2008.

[7] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[8] George EP Box and Kenneth B Wilson. On the experimental attainment of optimum conditions. *Journal of the royal statistical society: Series b (Methodological)*, 1951.

[9] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. In *Proceedings of the USENIX Security Symposium*, 2010.

[10] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015. https://ia.cr/2015/472.

[11] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[12] Cape privacy: Privacy & trust management for machine learning. https://capeprivacy.com/.

[13] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: programmable, efficient, and scalable secure two-party computation for machine learning. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[14] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Proceedings of the Annual Cryptology Conference (CRYPTO)*, 2012.

[15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - a framework for efficient mixed-protocol secure two-party computation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[16] Yevgeniy Dodis, Shai Halevi, Ron D Rothblum, and Daniel Wichs. Spooky encryption and its applications. In *Proceedings of the Annual International Cryptology Conference (CRYPTO)*, 2016.

[17] DualityTechnologies: Data encryption technology and secure collaboration. https://dualitytech.com/.

[18] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://ia.cr/2012/144.

[19] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C compiler for secure two-party computations. In *Proceedings of the International Conference on Compiler Construction (CC)*, 2014.

[20] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In *Proceedings of the International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 2015.

[21] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Privacy-preserving distributed linear regression on high-dimensional data. *Proceedings on Privacy Enhancing Technologies (PETS)*, 2017.

[22] EP George, J Stuart Hunter, William Gordon Hunter, Roma Bins, Kay Kirlin IV, and Destiny Carroll. *Statistics for experimenters: design, innovation, and discovery*. Wiley New York, NY, USA:, 2005.

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

[24] Simon F Goldsmith, Alex S Aiken, and Daniel S Wilkerson. Measuring empirical computational complexity. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2007.

[25] Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. Scalable multi-party private set-intersection. In *Proceedings of the IACR International Workshop on Public Key Cryptography (PKC)*, 2017.

[26] Wilko Henecka, Stefan K ögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. Tasty: tool for automating secure two-party computations. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2010.

[27] Wilko Henecka and Thomas Schneider. Faster secure two-party computation with less memory. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[28] William J Hill and William G Hunter. A review of response surface methodology: a literature survey. *Technometrics*, 1966.

[29] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2010.

[30] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the USENIX Security Symposium*, 2011.

[31] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the USENIX Security Symposium*, 2011.

[32] Inpher: Secret computing and privacy-preserving analytics. https://www.inpher.io/.

[33] Muhammad Ishaq, Ana Milanova, and Vassilis Zikas. Efficient mpc via program analysis: A framework for efficient optimal mixing. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[34] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. Gazelle: A low latency framework for secure neural network inference. In *Proceedings of the USENIX Security Symposium*, 2018.

[35] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.

[36] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: making spdz great again. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2018.

[37] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. Automatic protocol selection in secure two-party computations. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, 2014.

[38] Robert O Kuehl and RO Kuehl. Design of experiments: statistical principles of research design and analysis. 2000.

[39] Yehuda Lindell, Benny Pinkas, Nigel P Smart, and Avishay Yanai. Efficient constant-round multi-party computation combining BMR and SPDZ. *Journal of Cryptology*, 2019.

[40] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2015.

[41] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[42] Stuart Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 1982.

[43] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2017.

[44] Payman Mohassel and Peter Rindal. ABY3: a mixed protocol framework for machine learning. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.

[45] MPC alliance. https://www.mpcalliance.org/.

[46] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key FHE. In *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.

[47] Multi-party computation ceremonies in Zcash. https://z.cash/technology/paramgen/.

[48] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Proceedings of the Annual Cryptology Conference (CRYPTO)*, 2012.

[49] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved mixed-protocol secure two-party computation. Cryptology ePrint Archive, Report 2020/1225, 2020. https://eprint.iacr.org/2020/1225.

[50] Erman Pattuk, Murat Kantarcioglu, Huseyin Ulusoy, and Bradley Malin. Cheapsmc: A framework to minimize secure multiparty computation cost in the cloud. In *Proceedings of the IFIP Annual Conference on Data and Applications Security and Privacy (DBSec)*, 2016.

[51] Robin L Plackett and J Peter Burman. The design of optimum multifactorial experiments. *Biometrika*, 1946.

[52] Friedrich Pukelsheim. *Optimal design of experiments*. SIAM, 2006.

[53] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Knowledge inference for optimizing secure multi-party computation. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2014.

[54] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the Asia Conference on Computer and Communications Security (ASIACCS)*, 2018.

[55] SCALE-MAMBA. https://github.com/KULeuven-COSIC/SCALE-MAMBA.

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

[57] Scotiabank's chief risk officer on the state of anti–money laundering. https://mck.co/2ATh2IU, October 2019.

[58] Microsoft SEAL (release 3.4). https://github.com/Microsoft/SEAL, 2019.

[59] Shared machine learning: Ant financial's solution for data privacy. https://link.medium.com/CgDVD0mtbab.

[60] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-influence models for highly configurable systems. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2015.

[61] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P)*, 2015.

[62] Unbound tech. https://www.unboundtech.com/.

[63] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016.

[64] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[65] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the Symposium on Cloud Computing (SoCC)*, 2017.

[66] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the Symposium on Foundations of Computer Science (SFCS)*, 1986.

[67] Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. https://ia.cr/2015/1153.

[68] Tong Zhang. Adaptive forward-backward greedy algorithm for learning sparse representations. *IEEE Transactions on Information Theory*, 2011.

[69] Yihua Zhang, Aaron Steele, and Marina Blanton. Picco: A general-purpose compiler for private distributed computation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[70] Ruiyu Zhu, Darion Cassel, Amr Sabry, and Yan Huang. Nanopi: extreme-scale actively-secure multi-party computation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2018.