2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179397

Silph: A Framework for Scalable and Accurate Generation of Hybrid MPC Protocols

Edward Chen[†*], Jinhao Zhu[†*], Alex Ozdemir[‡], Riad S. Wahby[†], Fraser Brown[†], Wenting Zheng[†]
[†]Carnegie Mellon University [‡]Stanford University
{ejchen, jinhaoz, riad, fraserb, wenting}@cmu.edu, aozdemir@stanford.edu

***Abstract*—Many applications in finance and healthcare need access to data from multiple organizations. While these organizations can benefit from computing on their joint datasets, they often cannot share data with each other due to regulatory constraints and business competition. One way mutually distrusting parties can collaborate without sharing their data in the clear is to use secure multiparty computation (MPC). However, MPC's performance presents a serious obstacle for adoption as it is difficult for users who lack expertise in advanced cryptography to optimize. In this paper, we present Silph, a framework that can automatically compile a program written in a high-level language to an optimized, hybrid MPC protocol that mixes multiple MPC primitives securely and efficiently. Compared to prior works, our compilation speed is improved by up to $30000\times$. On various database analytics and machine learning workloads, the MPC protocols generated by Silph match or outperform prior work by up to $3.6\times$.**

1. Introduction

Advanced analytics and machine learning have become the dominant approach to solving many problems. To produce accurate results, these techniques require access to large amounts of high quality data. Unfortunately, obtaining sufficient volumes of data is challenging in many areas, such as healthcare and finance, because most high-value data is owned by and split across multiple organizations. While collaboration can enable organizations access to larger and higher quality datasets than what is available to any one organization, potential participants often own sensitive data that cannot be shared due to privacy concerns, business competition, and/or regulatory policies [30], [14].

Cryptographers have made significant progress towards a cryptographic approach to this problem called *secure multiparty computation* (MPC) [72], [7], [22], [34], [40], [41], [44], [51], [71]. 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. In fact, at the end of the computation, each party only learns the final result of the computation without uncovering any additional information about other parties' data or the intermediate results.

- [*] *Equal contribution.*

While promising in theory, MPC's concrete performance presents a serious obstacle for adoption as the techniques have high computation and communication costs. Additionally, no single MPC primitive performs the best for all workloads under all deployment settings. For example, arithmetic secret sharing [26], [23] can efficiently evaluate operations like matrix multiplication and convolutions, but is inefficient for non-linear operations like ReLU, sigmoid, and tanh. On the other hand, garbled circuits [72], [71] can efficiently evaluate non-linear operations but are inefficient for linear arithmetic operations. In many complex programs, both types of operations are often present, thus single-primitive protocols can be very inefficient.

In order to improve MPC performance, researchers have designed a number of hand-tuned, workload-specific protocols that combine multiple MPC primitives within a single application [10], [50], [52], [36], [59], [32], [31], [47]. For example, recent MPC protocols for neural network inference [36], [49], [43], [47] use a combination of homomorphic encryption, secret sharing, and garbled circuits. However, creating such protocols by hand is labor intensive and slow, making it difficult for users to quickly deploy MPC for their applications.

In response, MPC compilers aim to *automate* the process of generating workload-specific MPC protocols from user-provided programs that are written in a high-level language like C [16], [13], [35]. These tools focus on automatically mixing different MPC primitives in order to generate efficient hybrid MPC protocols. The optimization problem (called *hybrid protocol assignment*) aims to automatically assign code segments to MPC primitives and insert secure conversions between different primitives, while minimizing the predicted cost.

The fundamental challenge faced by such compilers is how to efficiently, scalably, and accurately generate workload-specific, hybrid protocols. Ultimately, each step in the computation must be assigned a primitive. Finding an optimal assignment might require *fine-grained reasoning*: considering each step individually. However, this is slow for large programs, so past compilers introduce many restrictions. Some tools use hardcoded heuristics and coarse-grained reasoning based on syntactic structures [16], [13]. Others place limits on the number of MPC primitives used within a single program [35].

In this paper, we present Silph, a new hybrid protocol

DOI 10.1109/SP46215.2023.00103

compiler for MPC that effectively Scales hybrid protocol assignment using ILP and other effective Heuristic-based tools. We exploit two insights in designing our techniques. First, heavyweight optimization machinery for protocol assignment is justified because MPC computation and communication cost varies widely over primitives; inappropriately selecting MPC primitives can cause orders of magnitude slowdown for complex applications [13]. Our second insight is that the protocol assignment problem can be expressed as an optimization problem. Unlike in plaintext programs, control flow in MPC programs is data-independent, making it easy to formulate well-posed optimization problems over data without making assumptions about program inputs. These two insights lead us to a compilation approach that uses heavyweight optimization tools to find accurate protocol assignments.

We begin by observing the protocol assignment optimization problem is a perfect fit for mixed integer linear programming (ILP) because it can model operation costs, assignment decisions, and conversion constraints. Therefore, formulating protocol assignment as an ILP problem and solving it using an ILP solver is a natural solution. We can derive the ILP constraints from a computation's intermediate representation (IR) term graph, allowing us to find a fine-grained protocol assignment for each step in a computation. We first design two new ILP formulations that have performance and accuracy tradeoffs. The first ILP formulation restricts each IR term to be assigned to a single MPC primitive. However, while this restrictive formulation can be solved faster, it is not optimal when the cost models have expensive conversion costs such that forcing a single assignment with conversions is worse than assigning a term to multiple primitives. Therefore, we design a second ILP formulation that removes this assumption by altering the first ILP's constraints.

Unfortunately, we quickly run into a performance obstacle by taking such a fine-grained approach to protocol assignment because ILPs are not easily scalable to large programs. Even worse, since our IR graph is equivalent to a low-level circuit representation, which exacerbates the scalability problem by greatly increasing the number of IR terms.

Therefore, we devise two techniques to improve scalability while maintaining assignment accuracy. First, we reduce the maximum input problem size to the ILP solver by partitioning a program's IR graph into local partitions. Unlike prior works, we find that a good partitioning does not necessarily correspond to syntactic structure specified at the input program level. Instead, we make the observation that partitions should be loosely connected to each other in order to minimize the cost of forced conversions across partitions. However, using optimal protocol assignments on local partitions may not yield a good global solution. Thus, we introduce a *partition mutation* heuristic that uses terms across partition boundaries to influence local assignments to better fit in the global assignment. Our first technique is as follows: we use graph partitioning to decompose the program circuit, followed by a two-level ILP approach (at the per-partition level and at the inter-partition level) to find an efficient assignment.

Our second technique is to use modular optimization. Since MPC programs are circuit representations of high-level programs, they also exhibit repeating sub-circuits that correspond to loops, functions, etc. While a repeating sub-circuit cannot be securely executed more than once with different inputs, its *protocol assignment* could still be reused across multiple executions. Therefore, we design techniques to reuse ILP protocol assignment work for such subgraph patterns. We first add a new function abstraction at the IR level to express high-level, user-provided modularity. We then adapt graph partitioning and partition mutation to work with functions. However, while it is possible to always reuse the same assignment for a function across *all* function calls, such a restriction may reduce the overall hybrid protocol's efficiency. Therefore, we design a new technique named call site similarity (CSS) analysis to enable multiple protocol assignments for the same function for calls within different contexts.

Silph is implemented on top of CirC [55], an extensible toolkit for circuit-based compilation. We evaluated Silph across 9 different workloads, including database analytics and machine learning. Our evaluation shows that our compilation pipeline is efficient, scalable, and produces accurate results. The compilation process is up to $30000\times$ faster and the generated hybrid protocols are up to $3.6\times$ faster on various database analytics and machine learning workloads.

2. Background

In this section we give background on multiparty computation, the circuit representations that these protocols require, and the way that compilers target those representations.

2.1. Secure multiparty computation (MPC)

A multiparty computation (MPC) protocol allows a set of mutually distrusting parties to collaboratively evaluate a function of all parties' private data. As one example, consider the classic Millionaires' Problem, where several competitive capitalists want to determine who among them is wealthiest without revealing their net worths. MPC solves this and similar problems, allowing P parties to compute a function f over their private inputs $x_1, \ldots, x_P$, without directly revealing x_i to any other parties.[1]

In order to execute a multiparty computation, the function f must be expressed as a *circuit*, i.e., a directed acyclic graph in which nodes (*gates*) have bounded fan-in and are labeled with an operation, edges (*wires*) are labeled with a value, and the graph has distinguished inputs and outputs; we say that a circuit is *satisfiable* if, for a given assignment

1. As is standard, we assume that the result is revealed to all parties, who thus learn whatever $f(x_1, \ldots, x_P)$ leaks about x_i. Dealing with this by careful design of f is outside the scope of this work. Moreover, we use the term MPC to refer generically to any $P \geq 2$.

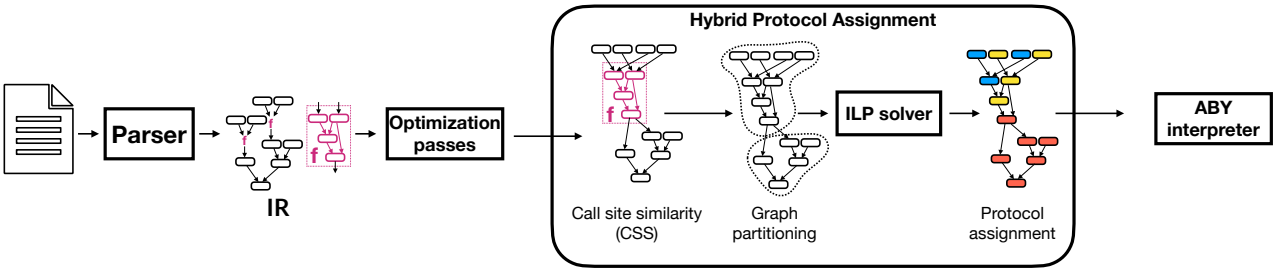


Figure 1: Silph workflow. A program written in a high-level language like C is first lexed and parsed into a circuit-based intermediate representation. Silph then applies IR optimizations and then passes the optimized IR to our hybrid protocol assignment framework. This will generate a corresponding protocol assignment, which is lowered to a concrete MPC execution that can be processed by our interpreter.

to its inputs and outputs, the out-edges of every node can be labeled with values consistent with that node's operation and input edges. There are two main MPC paradigms for generic computations—*arithmetic MPC* [8], [23] and *boolean MPC* [72], [71]—that correspond to different types of circuits. In arithmetic MPC primitives, wires take values from a finite field and node operations are addition and multiplication over that field. In boolean MPC primitives, wires take values from $\{0, 1\}$ and node operations are `AND` and `XOR` (in other words, arithmetic MPC over the field $\mathbb{F}_2$).

Hybrid MPC protocols [10], [50], [52], [36], [59], [32], [31], [47] combine two or more protocols, which may support different circuit types. At a high level, hybrid protocols work by running each MPC primitive in sequence, with connections among the protocols' circuits provided by special *conversion protocols* that securely translate wire values from one MPC primitive to another without revealing intermediate information.

2.2. Compilation for MPC

To execute an arbitrary program as an MPC, one must first encode that program as a circuit of the appropriate type. This is a notoriously tricky problem [65], essentially because MPC circuits have no notion of control flow or mutation of state. Such transformations are difficult and tedious to do by hand; this has inspired a long line of work on automatic compilation to MPC circuits [48], [45], [46], [73], [2], [58], [13], [35], [16], [74], [33], which we discuss in Section 9. Typically, such a compiler parses a source language (e.g., C) into an internal representation (e.g., an abstract syntax tree). The compiler then transforms that representation into a circuit-like representation (e.g., transforming state mutations, conditionals, and loops into circuits implementing that functionality), and lowers that circuit-like representation to a *target* representation (e.g., a protocol-specific library that provides primitives like gates).

We now briefly discuss the popular target framework Silph uses, called ABY, and give more information on compilation for hybrid protocols.

2.2.1. The ABY target. ABY [26] is a framework for constructing efficient hybrid MPC protocols for $P = 2$ parties built from three primitives: Arithmetic secret sharing, Boolean secret sharing, and Yao's garbled circuits. ABY provides efficient conversions among these protocols; e.g., `A2B` converts from an Arithmetic to a Boolean sharing.

ABY exposes a gate-oriented API that is higher-level than the gate representation in the underlying primitives. While MPC primitives only support `ADD`/`MUL` or `AND`/`XOR`, ABY also exposes slightly more complex operations `GT` (greater than) and `MUX`, as well as explicit conversion gates (e.g., `A2Y`, `Y2A`).

2.2.2. Compilation for hybrid protocols. Hybrid MPC protocols give the compiler an extra degree of freedom, namely, choosing which MPC primitive to use for each piece of the input program. This freedom makes it possible for the compiler to choose the most efficient MPC primitive for each part of the computation (e.g., field arithmetic is more efficiently expressed as an arithmetic circuit than a boolean circuit). Because conversions from one protocol to another incur overhead, however, protocol selection is a non-trivial optimization problem. We discuss prior work on the protocol selection problem in Section 9.

3. Overview

Silph is a compiler from high-level source languages to hybrid MPC circuits (see Figure 1 for the workflow). It is built on the CirC [55] toolkit for constructing circuit compilers, and inherits CirC's (standard) compilation pipeline: Silph lexes, parses, translates to a circuit-based intermediate representation (IR), optimizes that IR, and then lowers to a target representation—in Silph's case, the MPC-specific ABY backend. A key component in Silph is its hybrid protocol assignment framework, which does fine-grained assignment of each term in the IR to a concrete MPC primitive in order to minimize overall circuit execution time.

In the next sections, we first present Silph's threat model as background. Then, we walk through Silph's basic pipeline

in more detail, and finally give a high-level overview of Silph's hybrid protocol assignment phase.

3.1. Threat model

Silph currently supports ABY [26] as a backend, and ABY contains a set of MPC primitives that are secure in the two-party, semihonest setting. Silph is not limited to this threat model, though: it inherits the threat model of any supported backend, so adding a new backend will yield a new threat model.

It is important to ensure that the security of the overall protocol is still maintained when converting from one MPC primitive to another. In Silph, we assume that, for every pair of MPC primitives π_A and π_B, there exists a conversion protocol $\pi_{A,B}$ that can securely convert between the two protocols. In practice, ABY supports three distinct MPC primitives (A, B, and Y), as well as provably secure conversion protocols among them. Silph isn't limited to ABY's primitives, though; it is general enough to handle other primitives as long as they also support secure conversion protocols.

3.2. Silph's compilation pipeline

We describe the frontend languages Silph supports, the IR into which it translates source code, the optimizations it performs on IR, and its strategy for lowering IR to ABY.

3.2.1. Supported frontend languages. Since Silph is built on a compiler construction toolkit, adding support for a new language simply requires building a frontend from that language to Silph's IR. Silph currently supports two languages: the ZoKrates [27] language, originally designed for zero-knowledge proofs, and a subset of C. Silph's C subset includes booleans, signed and unsigned integers, structs, stack arrays, and pointers to (statically known) variables or arrays. It excludes while loops, recursion, goto, and the sizeof operator, and it neither detects undefined behavior nor tries to replicate particular implementation-defined behavior. Silph also supports the complete ZoKrates languages: variables, conditional expressions, statically bounded loops, booleans, fixed-width integers, finite field elements, structs, and field-element-indexed arrays.

3.2.2. Silph IR. Since Silph is built using the CirC compiler construction framework, it inherits CirC's intermediate representation, which is based on the SMT-LIB standard [4] and includes a rich set of operations over booleans, bit vectors, floating point numbers, finite fields, and arrays. However, CirC's IR isn't enough to support MPC. First, in an MPC there are multiple parties, each of whom has its own private inputs to the circuit that cannot be revealed to the other parties. To support a notion of parties and secrecy, Silph's IR includes metadata with each input to the `main` function.

The second IR modification—adding *function abstractions*—is essential for compiling large programs. Unfortunately, in an MPC circuit, there is no such thing as a function call; all calls must be inlined by the time the computation executes (§2.2). This inlining seriously inflates the size of the circuit, slowing down both compilation and optimization.

To address the function inlining problem, Silph includes function abstractions that compactly represents each function as a `Computation`, and each function call as a `Call`. To accommodate the function abstractions, Silph does not perform whole program optimization and instead optimizes per function (as many compilers do). Silph's backend ABY interpreter—discussed in a few paragraphs, Section 3.2.4—understands and interprets `Computation` and `Call` IR nodes.

3.2.3. IR Optimizations. Silph uses CirC's standard optimization passes (e.g., tuple elimination) and makes a few improvements to its constant folding (e.g., `x AND x` is replaced by `x`). Silph also includes an if-then-else (ITE) rewriting pass that eliminates unnecessary ITE terms produced by the compiler when it compiles conditional statements. Finally, we disable CirC's scalarization passes (e.g., linear scan) because they're expensive, and because our ABY interpreter supports non-scalars (§3.2.4).

3.2.4. Lowering to ABY using the ABY interpreter. After optimizations, the hybrid protocol assignment component (discussed next, §3.3) takes the IR graph as input and outputs a *share map*, an assignment of each IR graph node to a concrete share (i.e., A, B, or Y). Lowering from Silph's IR and the share map to ABY primitives is conceptually simple: there's a one-to-one mapping between almost every IR operation and some ABY primitive. Despite this conceptual simplicity, lowering introduces mechanical complexity. An early version of Silph used ABY's API to build a C file, and compiled that file to an executable using `gcc`—but compiling the generated code was prohibitively compute and memory intensive. A single ten-line input program could yield a hundred thousand lines of generated code that took around thirty minutes to compile.

To solve the ABY code generation problem, we created an interpreter for ABY bytecode. Silph's ABY interpreter takes as inputs files containing the inputs to the computation, the share map, and ABY bytecode (previously lowered from IR). It processes that bytecode and returns the result of the computation. Our ABY bytecode retains the IR's function abstraction, so when the interpreter reaches a `Call` term, it loads the corresponding function definition's bytecode (a `Computation` term) and re-wires the `Call`'s arguments and return value to point to the function definition. The interpreter also recognizes array selects and stores with a secret index, and expands them into ABY primitives. Finally, the interpreter inserts conversions where appropriate (e.g., if it encounters an addition between an input with share A and an input with share B).

3.3. Hybrid protocol assignment

The input to the protocol assignment component (Figure 1) is an IR graph; the protocol assignment phase tries to

answer the question, "how should each node in the graph be assigned to a concrete MPC primitive such that the overall circuit execution time is minimized?" We formulate this question as a mixed integer linear program (ILP), and use an ILP solver [29] to answer it. The ILP solver searches for a solution that minimizes the overall predicted execution cost of the circuit, modeling execution cost as the sum of each term's operation cost plus conversion costs. The solver returns a concrete assignment of MPC primitives to each node in the program IR graph.

In the rest of this paper, we present Silph's two ILP formulations for hybrid protocol assignment (§4). While ILP solvers are effective at finding protocol assignments, the problems they're attempting to solve are NP-complete. Solving very large problems, or finding assignments for very large circuits, is often intractable. To address this, Section 5 presents our strategy for scaling ILP by reducing the size of the input problem. Specifically, Silph automatically partitions the IR graph, and then (naively) uses an ILP solver to get protocol assignments for each partition. Since optimal protocol assignments for the local partitions don't necessarily translate to a globally optimal assignment, Silph uses a new *partition mutation* heuristic for exploring different partition assignments that are more globally optimal. Finally, in Section 6, we describe our *call site similarity* heuristic which uses function abstractions to enable modular optimizations and amortize parts of our hybrid protocol assignment approach.

4. ILP-based Protocol Assignment

In this section, we present our core approach to the protocol assignment problem. We express the optimization problem as a mixed integer linear program (ILP), and find protocol assignments using an ILP solver [29]. Compared to prior works that use hard-coded heuristics for assigning the correct primitives [13], [16], ILP allows Silph to generate accurate, fine-grained protocol assignments. In this section, we discuss two distinct ILP formulations for solving protocol assignments that have performance and accuracy tradeoffs. The next two sections (Sections 5 and 6) will present techniques to efficiently scale the ILP solving process.

4.1. Problem setup

We first explain protocol assignment's setup, as well as assumptions that we make. Since Silph supports ABY in the backend, each IR term can be assigned to three possible primitives: Arithmetic, Boolean, and Yao. We assume that the total cost of the program is a linear summation of the individual terms, i.e., the predicted cost of a program is the sum of each term's operation cost and the conversion costs.

4.2. ILP formulations

Defining the optimization problem as an ILP gives us a fine-grained protocol assignment solution that can mix

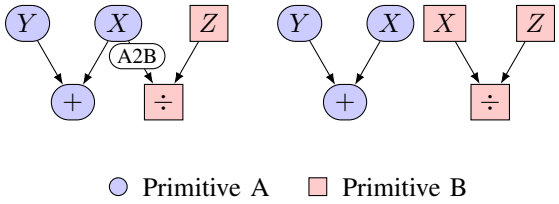


Figure 2: Left is a suboptimal assignment produced by the first ILP formulation; right is an optimal assignment produced by the second ILP formulation.

an arbitrary number of MPC primitives. Prior works like OPA [35] have explored formulating protocol assignment as an ILP, but their goal was to formulate it so that it can be relaxed it into a linear program. Unfortunately, in the process of doing so, OPA also had to restrict to mixing only two primitives at a time.

We first design a slightly simpler ILP formulation than that of OPA since we do not utilize LP relaxation. Let (t, s) denote a def-use IR term pair, i.e., s depends on t. Let $|t|$ denote t's bit length. Let a, b denote MPC primitives. We define two sets of boolean indicator variables. Let $T^{(t,a)} \in \{0, 1\}$ indicate whether a term t is evaluated using MPC primitive a. Let $C^{(t,a,b)} \in \{0, 1\}$ indicate whether term t needs to be converted from a to b. Let q_t^a indicate the cost of executing term t using primitive a, and $q_{|t|}^{a2b}$ as the cost of converting from primitive a to b. The concrete costs are experimentally identified, and we explain how to do this in Section 7. Note that the conversion cost depends on the size of the IR term, and larger sizes will have more expensive conversion costs.

The ILP formulation is

$$\sum T^{(t,a)} \cdot q_t^a + \sum C^{(t,a,b)} \cdot q_{|t|}^{a2b} \text{ subject to} \quad (1)$$

$$\forall t, \sum_a T^{(t,a)} \geq 1 \quad (2)$$

$$\forall a, b, \forall \text{ def-use pairs } (t, s),$$
$$C^{(t,a,b)} \geq T^{(t,a)} + T^{(s,b)} - 1 \quad (3)$$

The objective function in Equation (1) is the total cost of a program parameterized by $T^{(t,a)}$ and $C^{(t,a,b)}$, which are constrained by Equation (2) and Equation (3). Equation (2) constrains $T^{(t,a)}$ so that t must be assigned to at least one primitive. Equation (3) expresses the second constraint on the conversion variables $C^{(t,a,b)}$, which correctly adds a conversion cost if two connected terms are assigned to different primitives. Combined with Equation (2), each term t will only be assigned to a single MPC primitive since Equation (3) forces conversions for any pairwise distinct assignments (a, b) for (t, s), causing any term with multiple assignments to be strictly more costly than this term being only assigned to a single primitive.

The above ILP formulation is not optimal when a term is assigned to a single primitive such that it triggers expensive conversions to be inserted in the output. Take Figure 2 as an example. The term `X` is an input into both addition

and division terms. Let's assume that we have two MPC primitives, A and B. The cost of addition is 1 in A and 100 in B (addition prefers A). The cost of division is 200 in A and 2 in B (division prefers B). The conversion cost from A to B is 80, and B to A is 90. Since the ILP we presented forces X to be assigned to a single primitive, it would assign X to primitive A, the addition gate to operate in A, and division to operate in B, with a forced conversion from A to B. The total cost of doing so is 83. However, the optimal solution would be to duplicate X and represent it in both A and B. This duplication removes conversions, and the total cost is 3. Clearly, if conversions and non-preferred operator costs are all very expensive, then these forced conversions could greatly increase the overall cost and result in a suboptimal assignment.

Therefore, we design a second ILP formulation that improves upon the first formulation to allow a single term to be assigned to *multiple MPC primitives*. In order to formulate our design, we first make the observation that the original boolean indicator variables $T^{(t,a)}$ and $C^{(t,a,b)}$ are enough to express our new goals. Therefore, we can keep the same objective function and only modify the constraints.

If a single term t needs to be expressed in multiple MPC primitives, multiple $T^{(t,a)}$ variables can be assigned to 1 at the same time. Our current constraint on $T^{(t,a)}$ is already satisfactory because it is an inequality. The constraints on the conversion boolean indicators $C^{(t,a,b)}$, however, need to be changed in order to remove the restriction on single term assignment. As discussed earlier, Equation (3) will set a conversion boolean indicator $C^{(t,a,b)}$ variable to be true if the term pair can be assigned to primitives a and b. However, we need to modify it so that the least costly conversion is used when the term pair has multiple assignments.

We illustrate this using a backend that supports three different MPC primitives a, b, and c. For the def-use term pair (t, s), we can define 6 conversion variables. For each $C^{(t,x,y)}$, where $x \in \{a, b, c\}$, $y \in \{a, b, c\}$, and $x \neq y$, the inequality needs to be modified to relax the constraint. Let's take $C^{(t,a,b)}$ as an example. The conversion from a to b should be set to 0 if $T^{(t,b)} = 1$ (i.e., t is set to b, which is cheaper since there is no conversion needed). If $c2b$ is cheaper than $a2b$, and $T^{(t,c)} = 1$, then $C^{(t,a,b)}$ should also be set to 0. Otherwise, $C^{(t,a,b)}$ must be constrained to 1 since no other conversion is available. Therefore, the constraints should be modified to subtract the indicator variables of primitives where the conversions are cheaper, so that the conversion indicator variable can be assigned to 0. The new formulation is as follows:

$$\begin{aligned} &\forall t, \forall a \neq b, \forall s \text{ that depends on } t, \\ &\forall c_i \text{ such that } q^{(c_i 2b)} < q^{(a2b)} \\ &C^{(t,a,b)} \geq T^{(t,a)} + T^{(s,b)} - 1 - \sum_i T^{(t,c_i)} \end{aligned} \tag{4}$$

Finally, we note that the performance of these two ILP formulations could be quite different. Intuitively, the first formulation is more constrained since only one MPC primitive can be assigned to a term. This also means that the ILP is faster to execute, and we demonstrate the performance-accuracy tradeoff in our microbenchmarks in the evaluation section (Section 8.2.1).

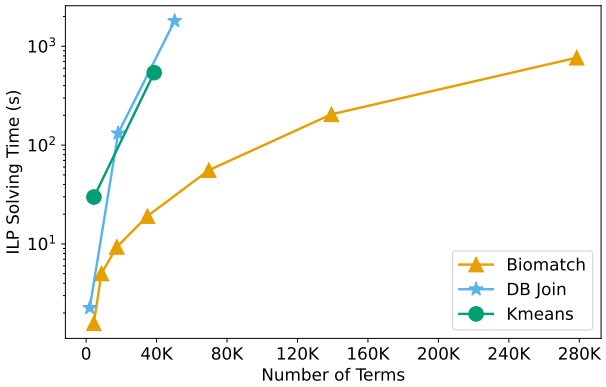


Figure 3: ILP solving time for three benchmarks of various input sizes: Biomatch: {256, 512, 1024, 2048, 4096, 8192, 16384}; DB Join: {10, 30, 50}; KMeans: {10, 100}.

5. Protocol Assignment with Partitioning

The previous section presents two ILP formulations for protocol assignment. However, this approach scales poorly with program size. To demonstrate this, we select three benchmarks from our test suites: Biomatch, DB Join and K-Means, and time the ILP-based assignment algorithm on these benchmarks with varying input sizes. Figure 3 shows the ILP solving time of these test cases and its relationship with the number of terms in each of the program. Clearly, ILP scales poorly. For example, increasing the input size of K-Means from 10 to 100 increases term count by 10×, but ILP solving time by up to 60× (depending on the source program). In this section, we present a technique to improve the scalability of ILP-based protocol assignment.

To improve the scalability of protocol assignment, we must reduce the size of the ILP problems we generate. Prior works like HyCC [13] use code partitioning. HyCC's code partitioning method is based on the *syntactic* structure (functions, loops, and arithmetic/boolean operations) of the input program. Within each partition, HyCC assigns a single MPC primitive to all nodes. It uses an exhaustive search to choose the best primitive for each partition. The partitioning approach here is fast but coarse-grained; furthermore, it is extremely sensitive to the syntactic structure of the input.

We take a different approach: our partitioning is based on the IR graph structure. We use a two-level ILP: first a per-partition ILP on (mutated) partitions, then a inter-partition ILP over the partition assignments.

5.1. Graph partitioning

In this section, we discuss our approach to code partitioning. There are two important metrics that are important

for producing a good partitioning. First, partitions may be well connected within, but should be only loosely connected to each other. Thus, graph cuts only cross a small number of edges, reducing the number (and cost) of forced conversions between partitions. Second, each partition should be small, as we plan to optimize each partition independently using our ILP solver. However, partitions cannot be too small: lest inter-partition conversion costs come to dominate within-partition costs.

Fortunately, these two metrics are common goals for graph partitioning algorithms. While graph partitioning is hard in the worst-case, we believe our graphs represent easy instances for graph partitioning algorithms. Our graphs are computation graphs; since most computations use far more time than space, our graphs are thus "long and straight". In general, graphs like social networks are harder to partition since they are better connected.

However, the above formulation can overestimate the impact of outgoing edges from a single node. We illustrate this with an example. Consider two situations in which a partitioning cuts n edges. In situation A, the n edges all have the same source. In situation B, the n edges all have difference sources. Graph partitioning algorithms view these two cuts as having equal cost. However, in a hybrid MPC protocol the output of any single node can be converted at most $k-1$ times, where k is the number of distinct primitives: 3 in Silph. Thus, situation A can induce at most 2 conversions, while situation B might induce n.

One way to solve this is to treat the IR graph as a hypergraph instead, where there is a single outgoing hyperedge connecting a node to its neighbors. We can then run a hypergraph partitioning algorithm, which weighs all hyperedges equally.

We use the KaHIP [60], [37] and KaHyPar [61], [38] libraries for graph and hypergraph partitioning respectively inside Silph. These libraries expose two parameters: p (number of partitions) and ϵ (imbalance). The library searches for a partitioning that minimizes the number of cut (hyper)edges, subject to the constraint that each partition's size is at most $(1+\epsilon)\lceil\frac{n}{p}\rceil$, where there are n nodes in the graph. Silph uses $\epsilon = 3$ (KaHIP's default) and sets p such that the average partition size is 8000. Increasing the average partition size slows down protocol assignment (since larger partitions slow down ILP solving) but may improve the final assignment.

5.2. Partition mutations and two-level ILP

Partitioning improves ILP solving time, but directly using per-partition assignments in the final global assignment is likely to increase MPC execution time. Intuitively, individually optimizing each partition without regard for neighboring partitions may induce expensive conversions between partitions. Our graph partitioning algorithm will alleviate some of these problems by identifying loosely connected partitions that have less data flow. However, the better assignments will also need to take into account the surrounding IR nodes in order to avoid overfitting to the partition's internal structure.

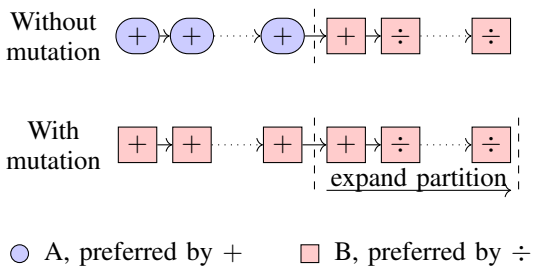
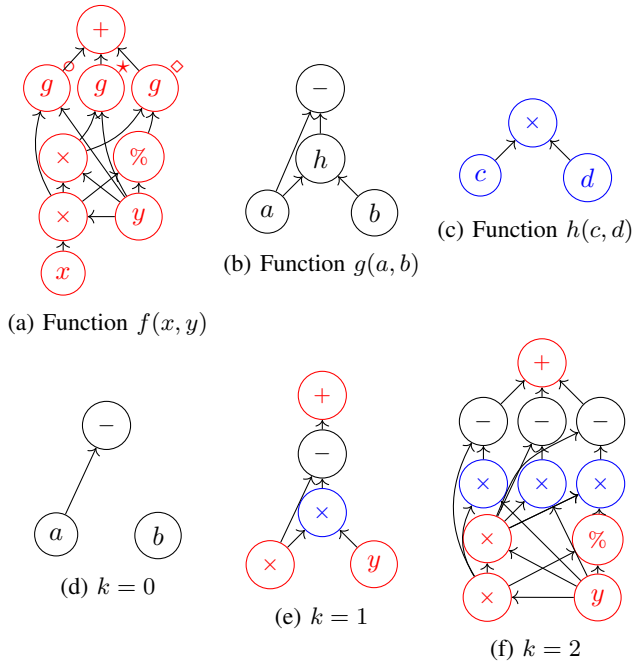


Figure 4: How mutations can help with protocol assignment

Therefore, we want to refine our partition-based optimization to better approximate running ILP over the entire graph. We design a new heuristic called *partition mutations* that allows us to change the partitions to explore more assignment options. Our idea is to mutate each partition by expanding it slightly, and run the ILP solver independently on the expanded partitions. We expand each partition by including not only the partition's nodes, but also the nodes that are immediately connected to the partition's edge nodes. Figure 4 shows a simple example of how partition expansion can help with certain scenarios. The two operators are $+$ (which prefers primitive A) and $\div$ (which prefers primitive B). The conversion cost between the two MPC primitives is relatively high and needs to be amortized over many operators. Thus, without mutations, the left partition will be uniformly assigned to A. However, if we are able to expand the left partition to include the division operators, then we will find an alternative cheaper protocol assignment that force the addition operators to use a less preferred assignment. This can save an extra expensive conversion cost.

Intuitively, there is a tradeoff in terms of how much to expand the partitions. If a partition is not expanded enough, then the ILP solver might not contain enough information from the surrounding nodes in order to make a good assignment for the nodes inside the partition. If a partition is expanded too much, then the ILP will overfit to the neighboring nodes and may not solve the local partition assignment problem well. Finding the correct expansion depth, however, is difficult as the right parameter will depend on the program's circuit structure. Therefore, we instead explore a range of expansion depths by setting a maximum depth that is constrained by the ILP solving time (called mutation level). Then, we will run ILP over each expanded partition to get multiple assignments, and finally execute a second, inter-partition ILP directly over these assignments. Our inter-partition ILP follows our per-partition ILP fairly closely, with the added constraint to handle conversion across partition boundaries.

Note that since this is a heuristic, and there are scenarios where partition expansion will not find the global optimum. Fortunately, this heuristic will do no worse than using a per-partition ILP assignment since we still use those assignments



(a) Function $f(x, y)$ (b) Function $g(a, b)$ (c) Function $h(c, d)$

(d) $k = 0$ (e) $k = 1$ (f) $k = 2$

Figure 5: Function mutations for call stack $(\star, g)$

as options into the second round of ILP.

6. Function-based Protocol Assignment

The previous section presents techniques for partitioning an IR graph to reduce the maximum input size to the ILP solver. In this section, we explore an alternative, complementary technique to efficiently and accurately take advantage of modular optimization.

So far, we have assumed that the protocol assignment phase has access to a single, unified IR graph. However, inlining all functions not only lead to excessively large IR graphs, but also introduce longer protocol assignment times where the ILP have to repeatedly solve for the same sub-circuit structures. At the same time, not inlining functions will instead give the protocol assignment framework a *set* of IR graphs that are indexed by function names. Adapting our partitioning and mutation techniques to graphs with function representation is non-trivial: we need to reuse ILP solving work to further scale to larger programs, but also preserve assignment accuracy. In this section, we discuss how to generate protocol assignments for MPC programs with functions.

Example Figures 5a through 5c show our running example for this section. The entry function is f and there are two other functions, g and h. g is called three times within function f, while h is called once within g.

6.1. Straw man

A naive protocol assignment approach is to independently run protocol assignment for each function, ignoring the cost of `Call` terms. The limitation is obvious: the cost of converting between primitives across function boundaries is not accounted for. There are two classes of ignored conversion costs. First, the cost of computing a function's output into the forms needed by its dependencies. Second, the cost of converting a function call's actual arguments to the form expected for its formal arguments.

Inlining all functions can account for these costs but scales poorly with overall program size. Therefore, we want to design techniques that allow us to take advantage of modular optimization without ignoring the cost of function calls.

We depart from the naive approach in three ways. First, instead of running protocol assignment once for each function, we run it for each calling context that a function occurs in. Second, we use partition mutations to account for conversion costs across function boundaries (§6.2). Third, we apply the result of protocol assignment for one calling context to similar calling contexts (§6.3).

6.2. Partitioning and mutations with functions

The conversion costs around a function depend on both its callers and callees. For any function f, the set of callees is fixed, but there may be many callers and many calling contexts for f. Thus, we first consider each calling context separately during protocol assignment.

Let f_d be a function called at depth d. Let $f_1, \ldots, f_{d-1}$ be the sequence of parent functions for f_d with call terms $t_1, \ldots, t_{d-1}$. Thus, f_1 is the entry function, and for $i < d$, term t_i in f_i is a call of f_{i+1}. We call the tuple $(t_1, \ldots, t_{d-1}, f_d)$ a *call stack*. Each call stack uniquely identifies a copy of f_d in the inlined IR graph. For a call stack $(t_1, \ldots, t_{d-1}, f_d)$ and distance parameter k, we generate a mutation f' that contains all nodes at distances $\leq k$ from f_d in the inlined IR graph. Note that f' certainly contains nodes from f_{d-1} and nodes from the callees of f_d. However, it may also contain nodes from *other* functions, such as callees of f_{d-1} whose call-site is close to that of f_d. With careful indexing, f'_d is constructible in time and space proportional to its number of nodes; the inlined IR graph need not be fully materialized.

Each function mutation f'_d may be further split. If its size exceeds the partition size parameter (§9), then we use graph partitioning to further decompose it, and we use the aforementioned mutation generation algorithm to generate mutations that reach across these sub-partition boundaries.

As before, Silph then performs ILP-based assignment on all mutations. The resulting assignments are restricted to the nodes that that mutation is uniquely responsible for (all of f_d, if the partition size was not exceeded). Then (as before) we construct a inter-partition ILP to optimally choose between the different assignments for each partition.

Example Figure 5 shows an example of function mutations. The function g has three call stacks: $(\circ, g)$, $(\star, g)$, and $(\diamond, g)$. Figure 5d shows the $k = 0$ mutation for stack $(\star, g)$. Figure 5e shows the $k = 1$ mutation for the same stack; this

mutation contains a node of the callee h (in blue) and also nodes of the caller f (in red). Figure 5f show the $k = 2$ mutation; all nodes of the inlined computation except x are now included.

6.3. Call site similarity analysis

The above approach never materializes the inlined IR graph in memory, but it ultimately runs ILP-based protocol assignment for the whole graph, piece by piece. In particular, for a function f called in many contexts, ILP-based assignment runs on mutations of f separately for all calling contexts. This is expensive, and it is unneeded if different calling contexts ultimately require the same kinds of conversions to be performed. To reduce the number of ILP invocations, we use *call-site similarity analysis*.

This analysis aims to cluster different call stacks for the same function into groups that be analyzed together. For each call stack for f, consider the actual parameters to f and the immediate dependents of its output. Two stacks are grouped if the operators of the actual parameters and immediate dependents are equal. For the actual parameters, the sequence of operators must be equal. Since the immediate dependents have no order, it is the *multisets* of their operators that must be equal.

For each group of similar call stacks, mutations and assignments are computed only once, for an arbitrarily selected call stack. The result is re-used for all call stacks in the group when constructing the global protocol assignment.

Example Recall that in Figure 5 there are three call stacks for g: $(\circ, g)$, $(\star, g)$, and $(\diamond, g)$. The output from each call to g is used by the same node. For stacks $(\circ, g)$ and $(\star, g)$, the input operators are a multiplication and a variable. Thus, we only analyze mutations for *one* of these stacks. Since the input operators for $(\diamond, g)$ are a multiplication and a remainder, that stack is analyzed separately.

7. Implementation

We implement Silph using the CirC compiler toolkit [55] and the ABY 2PC framework [26]. Our implementation is ≈10.3k lines of Rust and C++ code. The latest version of Silph can be found at https://github.com/edwjchen/Silph.

Although there exists more advanced hybrid MPC frameworks [57], [12], [2], (e.g., MOTION), we decided to target ABY primarily because our baselines, HyCC and OPA, both use ABY as their target backend. We believe that Silph can support these other frameworks with minimal development effort since they share a similar interface with ABY.

7.1. Protocol Assignment

To evaluate our techniques, we implemented three different protocol assignment schemes.

Global ILP (G-ILP) In this configuration, Silph uses ideas from Section 4. It inlines all function calls and then performs a single ILP-based protocol assignment for the full computation graph.

Two-level ILP (T-ILP) This configuration implements Silph's two-level ILP scheme from Section 5. Silph inlines all functions, partitions the complete IR graph, and solves a per-partition ILP. Silph then runs the partition mutation heuristic and finally finds a global assignment using the inter-partition ILP.

Call Site Similarity ILP (C-ILP) This configuration implements our designs from both Section 5 and Section 6. Functions are not inlined. Rather, we first partition the IR graph on function boundaries and append each function with the terms in the surrounding calling context. We then use call site similarity analysis to group similar function call stacks before running a per-function ILP and constructing a global assignment.

7.2. Cost Model Generation

In Silph, the ILP solver uses a cost model to find the most efficient protocol assignment for a given computation. The cost model is a one-to-one mapping between an ABY operation (e.g., Arithmetic addition or `A2B` conversion) and its expected runtime. To generate a suitable cost model, we measure the execution time of randomly generated circuits produced from CostCO [28], an automatic MPC cost modeling framework. The cost consists of both the setup and online phases of an ABY operation. Each unique execution environment requires a corresponding cost model; thus, we generated two cost models, one for the LAN setting and one for the WAN setting (as defined in §8.1).

One challenge we faced when generating our cost model was incorporating the interactive latency of an operation since each primitive can have a different number of communication rounds for a given operation. For example, a multiplication expressed in arithmetic secret sharing takes 1 communication round whereas Yao's garbled circuits could take a constant number of communication rounds. Furthermore, operations that are not data dependent can be batched within the same communication round, thereby amortizing the communication cost of the operation. To account for the effects of latency, we heuristically model the cost of an ABY operation as (operation cost without interaction) + `k` × (interactive latency of the operation), where `k` is an approximation of the number of non-batched, interactive operations. To estimate `k`, we used this simple heuristic: `k` = the data dependent depth of interactive operations / the total number of interactive operations. While this is a limitation in our system, we find that our heuristics estimate the cost model well in practice. We leave exploring and integrating better cost models for future work.

8. Evaluation

This section presents our evaluation of Silph. First we describe our evaluation setup, followed by detailed microbenchmarks evaluating the tradeoffs of our heuristic-based techniques. Next, we present end-to-end comparisons

	ILP 1		ILP 2	
Benchmark	**Pred. RT**	**ILP Time**	**Pred. RT**	**ILP Time**
GCD	322.7	0.02	322.7	0.03
Histogram	10194.2	25.34	10130.6	129.77
Biomatch	3299.3	1.72	3299.3	27.63
K-Means	5338.2	28.48	5052.7	18.36
Gauss	532.4	1.52	527.7	0.54
DB Merge	661.0	0.38	659.5	1.7
DB Join	16445.4	23.87	16323.4	89.96

TABLE 1: ILP Performance using LAN cost model

	ILP 1		ILP 2	
Benchmark	**Pred. RT**	**ILP Time**	**Pred. RT**	**ILP Time**
GCD	253.0	0.03	253.0	0.01
Histogram	566600.0	7.078	15708.0	4.99
Biomatch	156416.0	4.52	3328.0	0.38
K-Means	490200.0	1.61	4400.0	0.45
Gauss	16788.0	0.03	343.0	0.02
DB Merge	1897.0	0.24	1000.0	0.13
DB Join	898006.0	10.84	18035.0	5.3

TABLE 2: ILP Performance using synthetic cost model

against our two baselines, HyCC and OPA. We conclude our evaluation with a conceptual and empirical comparison of Silph against the relaxed LP protocol assignment approach used in OPA.

8.1. Evaluation Setup

All benchmarks were compiled on AWS EC2 using a single c6a.16xlarge instance (64 cores, 128 GB). The benchmarks were then evaluated using two r5.xlarge instances (4 cores, 32 GB) in both LAN (both instances in `us-east-2`) and WAN (one instance in `us-east-2` and another in `us-west-2`) environments. We did not limit the network of our evaluation instances, and thus each r5.xlarge instance had up to 10Gbps of upload and download bandwidth.

For our benchmarks, we used experiments that have previously been used to evaluate other MPC protocols and compilers. These benchmarks include common workloads used in database analytics and machine learning. The benchmarks are as follows. OPA introduced two new benchmarks: Greatest common denominator and Histogram (bucketing a list of points into a histogram). The remaining benchmarks were used to evaluate HyCC: Gaussian elimination, Biometric matching (minimum euclidean distance between a point and a database of points), DB Merge, DB Join, K-Means clustering, and convolutional neural networks from MiniONN and CryptoNets.

To partition our IR graph, we used KaHIP and KaHyPar, a graph and hypergraph partitioning library, respectively. Each of these libraries had a configurable set of parameters to tune the partitioning mode, speed, and quality. For KaHIP, we used the `KaFFPa` algorithm with the `fast` variant to prioritize partitioning time. For KaHyPar, we used the default `direct` partitioning mode, `imbalance` factor of 3, and `time-limit` of 3600. Additionally, we used the `cut` objective function and the `cut_kKaHyPar_sea20.ini` preset file in order to minimize the number of inter-partition edges.

8.2. Microbenchmarks

In order to isolate the effects of our techniques, we conducted microbenchmarks for our two ILP formulations: partitioning and mutation heuristics, and modular optimization with call site similarity.

8.2.1. ILP formulations. Our first microbenchmark compares the solving time and accuracy of our two ILP formulations: ILP1, where a term can be assigned to only one MPC primitive (Equation (3)), and ILP2, where a term can be assigned to multiple MPC primitives (Equation (4)). To compare the assignment quality from each ILP formulation, we use the ILP solver's output, which represents the predicted runtime of the target computation.

Table 1 shows the results of evaluating both ILP formulations on several benchmarks using our LAN cost model. As hypothesized, our ILP2 formulation is able to find better predicted runtimes in 5 out of 7 benchmarks because this formulation can assign a single term multiple assignments, thereby producing more fine-grain assignments. However, ILP2 incurs a higher ILP solving time because the problem's search space is increased. For these benchmarks, we find the benefits from improved predicted runtime performance are insufficient to offset the higher ILP solving time cost. This is because the conversion costs in our LAN cost model are of the same magnitude as operation costs. Therefore, we use ILP1 over ILP2 for our macrobenchmarks.

To demonstrate the possible improvements of using ILP2, we conducted another microbenchmark using a synthetic cost model with inflated conversion costs. This cost model contains only two primitives, A and B. The costs of linear operations (`ADD, MUL, SUB`) are set to 1 using A and 300 using B; all other costs were set to the opposite, 300 using A and 1 using B. The conversion cost is set to 1000 in either direction. Table 2 shows the results of both ILP formulations on the same benchmarks using this synthetic model. On 6 of the benchmarks, ILP2 was able to achieve $1.8-111\times$ speed-up in predicted runtime compared to ILP1. In GCD, all operations used in this benchmark prefer Y. Since no conversions are necessary, ILP1 outputs the same predicted time as ILP2.

8.2.2. Partitioning and mutation. Next, we evaluate the effects of graph and hypergraph partitioning, with and without partition mutations, on protocol assignment accuracy and efficiency. Our first goal is to discern the baseline protocol assignment accuracy and efficiency tradeoffs between our two proposed graph and hypergraph partitioners, KaHIP and KaHyPar. Our second goal is to show that mutations can improve the overall quality of protocol assignments.

We design our microbenchmark as follows. For each application, we run both partitioners using a partition size of 1000 and a varying mutation level of (0, 1, 2, 4). We then

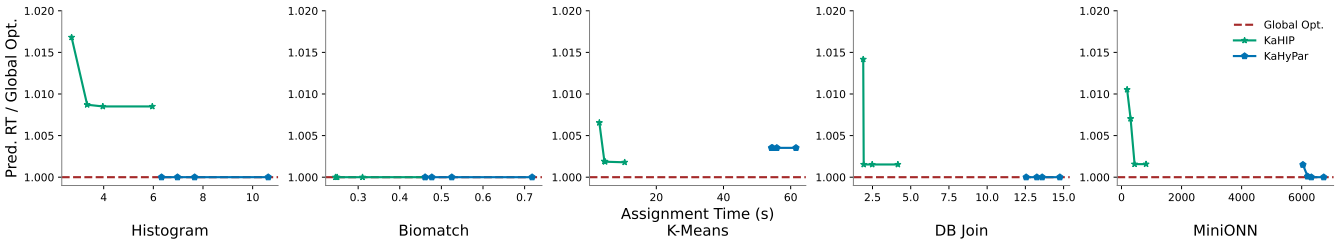


Figure 6: Predicted runtime (Pred. RT) given by ILP solver using different graph partitioners and mutation levels. Global Opt. is the optimal predicted runtime given by G-ILP. Assignment time includes partitioning and ILP solving time.

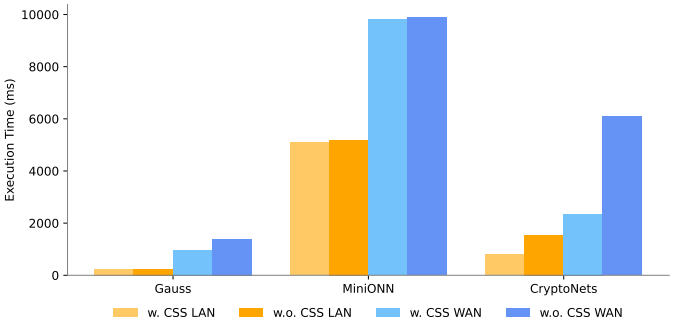


Figure 7: A comparison between with CSS and without CSS

compare the accuracy (predicted runtime of T-ILP/ predicted runtime of G-ILP) and efficiency (assignment time) of each partitioner. Figure 6 shows the results of this experiment on a subset of the benchmarks. The green and blue lines represent the protocol assignment accuracy from using different graph partitioning libraries. Each point on a line represents a mutation level from 0 (leftmost) to 4 (rightmost). The red dotted line represents the baseline predicted runtime of G-ILP, which has a constant value of 1.

To compare the partition quality of each graph partitioner, we first analyze the predicted runtimes with a mutation level of 0. Our microbenchmark shows that KaHyPar achieves close to the G-ILP baseline on all 5 benchmarks, whereas the predicted runtime from KaHIP can be much slower. Although KaHyPar gives better quality partitions, which reduces extraneous conversion costs across inter-partition boundaries, it takes approximately $2.5-30\times$ longer to partition on larger applications like DB Join and MiniONN. By introducing partition mutations, we are able to greatly improve the predicted runtimes of T-ILP with KaHIP. Using KaHip with 2 to 4 mutations, we achieve close to the G-ILP predicted runtime baseline on DB Join and CryptoNets, in much less time compared to KaHyPar. Therefore, we decide to use KaHIP with a mutation level of 2 in our macrobenchmarks.

8.2.3. Call site similarity. For our last microbenchmark, we evaluate call site similarity by comparing C-ILP against a naive per-function ILP baseline that restricts a function to a single set of assignments with no mutations. Figure 7 shows the execution runtimes on three representative applications. C-ILP achieves slightly better results in both LAN and WAN for Gauss and MiniONN, and a $2\times$ runtime speedup on CryptoNets. The reason for this performance gain is attributed to better protocol assignments by exploring the contexts outside of a function boundary. In CryptoNets, the activation layer uses `square` as the activation function. This function is represented by a single `MUL` operation. Under the WAN cost model, the operation cost of `MUL` is cheapest using Yao's garbled circuits. Without CSS, the ILP formulation cannot explore the contexts surrounding a function and will assign this `MUL` operation to Yao. However, the input wires to `MUL` come from an `ADD` assigned to Arithmetic from the parent function. Therefore, a conversion from `Y2A` is forced for every call to `square`, incurring a large slow-down during circuit execution. When CSS is enabled, the context of the activation function is added to the ILP formulation, thereby assigning `MUL` to be Arithmetic.

8.3. Macrobenchmarks

In this section, we compare the compilation and circuit evaluation performance of Silph to that of HyCC. Similar to Silph, HyCC's implementation compiles C programs into hybrid MPC protocols that target ABY. HyCC partitions code based on syntactic structure (functions, loops, and arithmetic/boolean operations), and compiles each partition into Arithmetic (if possible), Boolean, and Yao. HyCC then provides two ways to generate protocol assignments. The first is a heuristics-based approach (HyCC-H) that will greedily prioritize select MPC primitives and assign a partition the highest priority primitive available. In Figures 9a and 9b, we present the fastest HyCC-H directly using their execution runtimes. The second is an exhaustive search algorithm (HyCC-PSO i.e., protocol selection optimized) that uses a cost model to minimize the total cost of assignments for each partition. Lastly, HyCC contains logical minimization techniques to further optimize their boolean circuits at a gate level.

The evaluation parameters for each system are as follows. For Silph, we set a partition size of 8000, a mutation level of 2, and evaluated all three aforementioned selection schemes from 7.1. Through empirical testing, we found that for all test cases, setting a partition size of 8000 was

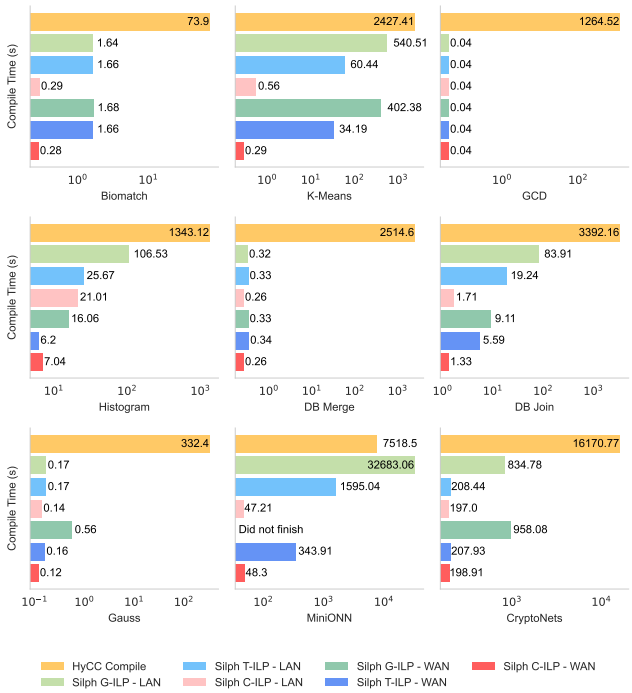


Figure 8: Benchmark compile times results comparing HyCC with Silph

sufficient in balancing partitioning time, ILP solving time, and memory usages. From our microbenchmarks 8.2.2, we discovered these mutation parameters were sufficient in finding a predicted runtime close to that of G-ILP.

For HyCC, we set the circuit minimization time to 600 seconds (default) and compiled with the `--all-variants` flag in order to generate hybrid MPC protocols. For Gaussian, specifically, the minimization time was set to 9 instead of 600 because compilation of the circuit did not finish when minimization time was ≥ 10. Although HyCC has an additional `--outline` flag for arithmetic decomposition (splitting groups of arithmetic computations into separate, smaller functions) we found that the implementation was limited.

However, we believe our benchmarks aforementioned in 8.1 are representative of HyCC's best efforts since the benchmarks from HyCC were written with arithmetic decomposition in mind. For example, in their MiniONN benchmark, the nonlinear ReLU function is already decomposed into a separate function from the linear convolution functions. We did not decompose the benchmarks from OPA because most of the computation is nonlinear, which favors the boolean circuit optimizations in HyCC.

Compilation and protocol assignment. Figure 8 shows the total compilation times (including protocol assignment) of all benchmarks for both Silph and HyCC. The first bar represents the total compilation time for HyCC, and the remaining bars represent the total compilation for Silph running G-ILP, T-ILP, and C-ILP in LAN and WAN settings. In most cases, Silph is at least an order of magnitude faster in compilation time, ranging from $12\times$ faster in Histogram using Silph's G-ILP (LAN) to over $30000\times$ faster in GCD using Silph's T-ILP (LAN). Notably, our T-ILP and C-ILP techniques are always faster compared to HyCC; C-ILP is consistently orders of magnitude faster.

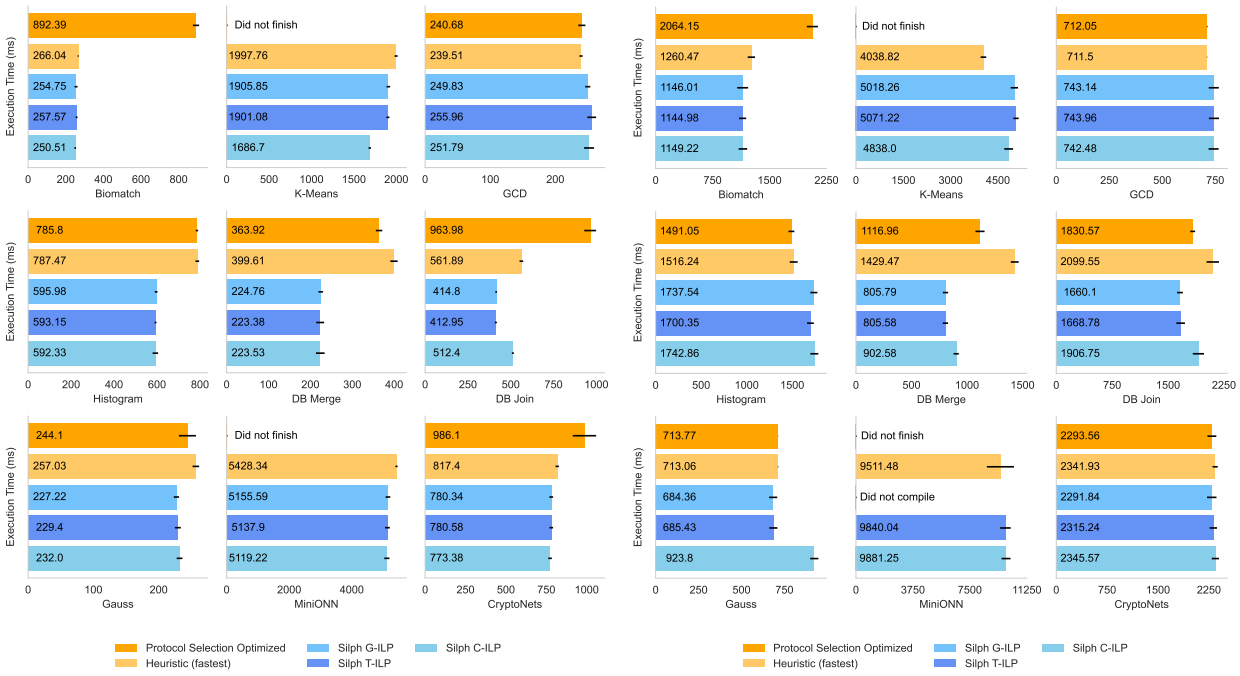
The reason for HyCC's slow compilation time in GCD is due to their time-based, boolean gate-level minimization pass. Although there is only a single small function in this benchmark, this function is compiled to both a Boolean and a Yao circuit, each circuit taking the full 600 seconds to optimize. If we were to compare against HyCC's compilation time with no minimization time, CryptoNets would take HyCC $\approx$12483 seconds to compile. This is because their protocol assignment algorithm is an exhaustive search that compares all possible share combinations of each function. In this case, Silph is $65\times$ faster in total compilation time compared to HyCC.

There are a few notable outliers in Figure 8, specifically in the MiniONN benchmark. When solving the G-ILP formulation (LAN), Silph takes around 10 hours to find the protocol assignment because the ILP solver needs to solve for $\approx$1,400,000 terms. In the WAN setting, a segmentation fault occurred in the ILP solver's library after $\approx$5000 seconds of solving. Additionally, HyCC-PSO was unable to finish for MiniONN. With constant propagation, the MiniONN benchmark has 22 unique functions, each having 3 possible share type assignments. Thus, HyCC-PSO has 3^{22} combinations to consider. After 24 hours, the exhaustive search algorithm only evaluated 125,670 combinations.

Execution. Figures 9a and 9b show the average execution time (across 10 runs) and standard deviation (black lines) of all benchmarks in both LAN and WAN settings. Aside from the MiniONN benchmark that either cannot compile (G-ILP) or finish selecting (HYCC-PSO), KMeans for HyCC-PSO also does not finish. This is because the HyCC circuit runs out of memory when trying to execute.

Silph is faster on 8 / 9 LAN benchmarks and 6 / 9 WAN benchmarks. Silph achieves a $3.6\times$ speed up compared to HyCC-PSO in Biometric matching (LAN) and a $1.8\times$ speed up compared to HyCC-H in DB Merge (WAN). We attribute our performance gains to having better, more fine-grain protocol assignments compared to HyCC. This is made evident by closely examining the protocol assignments for DB Merge. Although Table 3 shows that both systems found A+Y assignment, HyCC assigned the Mean and Variance functions in this benchmark to `Y` because the functions ended with a division. On the other hand, Silph was able to assign Arithmetic shares to a majority of Mean and Variance, resulting in a $1.8\times$ performance improvement for Silph.

However, Silph is slightly slower than HyCC for GCD (LAN/WAN), K-Means (WAN), and Histogram (WAN). By analyzing the protocol assignments in Table 3, we see that Silph finds the same protocol assignments as HyCC. In particular, both systems find only `Y` assignments for GCD. Therefore, the difference in performance is not due to assignment quality, but rather differences in the underlying circuit. We attribute HyCC performance gains to their



(a) LAN (b) WAN

Figure 9: Benchmark circuit execution times results comparing Hycc with Silph

Application	HyCC PSO		HyCC Heuristics (fastest)		Silph G-ILP		Silph T-ILP		Silph C-ILP	
	LAN	WAN	LAN	WAN	LAN	WAN	LAN	WAN	LAN	WAN
Biomatch	Y	Y	A+Y	A+Y	A+Y	A+Y	A+Y	A+Y	A+Y	A+Y
K-Means	Y	Y	A+Y	A+Y	A+B+Y	A+Y	A+B+Y	A+Y	A+B+Y	A+Y
GCD	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Histogram	Y	Y	Y	Y	A+Y	A+Y	A+Y	A+Y	A+Y	A+Y
DB Merge	A+Y	A+Y	A+Y	A+Y	A+Y	A+Y	A+Y	A+Y	A+Y	A+Y
DB Join	Y	Y	A+Y	A+Y	A+Y	A+Y	A+Y	A+Y	A+Y	A+Y
Gauss	Y	Y	Y	Y	Y	Y	Y	Y	Y	A+Y
MiniONN	-	-	A+Y	A+Y	A+Y	-	A+Y	A+Y	A+Y	A+Y
CryptoNets	A	A	A	A	A	A	A	A	A	A

TABLE 3: Protocol assignments between HyCC and Silph in LAN and WAN setting. Filled cells represent the selection scheme with the fastest execution time (average across 10 runs).

boolean gate-level compilation pipeline and optimizations, which Silph lacks.

Finally, our evaluation shows that C-ILP, even with the fastest compile times from Figure 8, can still achieve reasonable performance results compared to the other two selection schemes in Silph. Aside for DB Join, C-ILP is able to find a protocol assignment that results in near identical circuit execution times compared to G-ILP and T-ILP.

8.4. Comparison to OPA

In this section, we present the conceptual and empirical tradeoffs of the protocol assignment techniques used in OPA to that used in Silph. We evaluate OPA by integrating their LP relaxation into Silph's compilation pipeline. Table 4 shows the protocol assignment solving time and circuit execution results of OPA's LP relaxation compared to our G-ILP and C-ILP techniques. All experiments were evaluated in the LAN setting.

For the protocol assignment process, OPA uses a global relaxed LP approach that is restricted to at-most 2 primitives. When solving for the optimal protocol assignment using ABY (3 primitives), OPA executes their LP formulation for every pair of unique primitives (`AB, BY, AY`) and selects the assignment with the best predicted runtime. From Table 4, we see that OPA's approach scales better compared to G-ILP, improving protocol assignment solving time by up to 300× (in K-Means). This is because their LP relaxation and restriction to at-most 2 primitives greatly reduces the search space that the LP solver needs to solve for.

	LP Solving			Circuit Execution		
Benchmark	**G-ILP**	**C-ILP**	**OPA**$_{Total}$	**G-ILP**	**C-ILP**	**OPA**$_{Best}$
Biomatch	1.896	0.094	0.283	0.254	0.250	0.263
K-Means	1195.223	0.452	4.570	1.905	1.686	1.956
GCD	0.019	0.018	0.014	0.249	0.251	0.253
Histogram	105.233	18.067	2.846	0.596	0.592	0.588
DB Merge	0.212	0.144	0.064	0.225	0.223	0.219
Db Join	82.297	0.475	1.109	0.414	0.512	0.453
Gauss	0.094	0.057	0.014	0.227	0.232	0.228
MiniONN	32633.4	4.413	5890.65	5.156	5.119	5.207
CryptoNets	642.483	7.499	216.703	0.780	0.773	0.795

TABLE 4: Comparison of the protocol assignment and circuit execution times using G-ILP, C-ILP, and LP relaxation from OPA. OPA$_{Total}$ represents the *total* solving time for all combinations of two primitives using ABY, (`AB`, `AY`, `BY`). OPA$_{Best}$ represents the *best* execution circuit time from the 3 combinations.

On the other hand, Silph's ILP formulation can support an arbitrary number of primitives. Therefore, Silph would be able to find a more optimal protocol assignment if a benchmark used more than 2 primitives. When one considers function-based protocol assignment using C-ILP, the relationship between Silph and OPA is more complex. Although C-ILP is based on ILPs, it is faster than OPA's global LP, because each function ILP is quite small. Silph can also employ its partition mutations heuristic to further decompose large functions into smaller ILPs. The accuracy of C-ILP's protocol assignments (i.e., runtime performance) is also comparable to OPA's LP, differing by at most 0.1 seconds.

We leave the question of combining the LP-based approach of OPA with the multi-level approach of Silph to future work.

9. Related work

MPC compilers. Silph is part of a line of works on compilers for secure multiparty computation [48], [45], [46], [73], [2], [58], [13], [35], [16], [74], [33]. Many (early) works in this area focused on compiling a high-level program to an individual MPC primitive, while more recent works focused on generating workload-specific hybrid MPC protocols.

HyCC [13] compiles C programs to hybrid MPC protocols. It partitions programs based on syntactic structure (i.e., functions, loops, arithmetic expressions). All computations in a partition are evaluated using the same primitive. In contrast, Silph lifts these restrictions by analyzing and optimizing based on the IR graph, thus making our generated protocols less sensitive to the syntactic structure of the input program. We also produce fine-grained assignments by allowing each partition to be assigned to multiple primitives.

EzPC [16] compiles high-level programs to hybrid two-party computations (2PC). It employs a specialized, heuristics-based algorithm that partitions a program based on arithmetic and boolean operations, and assigns a single 2PC primitive to each partition (similar to HyCC). In comparison, Silph can perform protocol assignment for any set of MPC primitives based on a cost model instead of using heuristics to determine assignments.

OPA [35] is another work that tackles the protocol assignment problem from a theoretical angle. It models the problem as an ILP, which is carefully constructed to achieve a nice theoretical result: it is equivalent to its LP relaxation when there are only two primitives. Silph's first ILP formulation simplifies OPA's formulation at the expense of LP-relaxation equivalence. OPA's ILP has the same restriction as Silph's first ILP in that each computation unit is only assigned to one primitive. Silph's second ILP formulation generalizes the optimization problem allows computation units to be assigned to multiple primitives.

Viaduct [1] compiles programs with confidentiality and integrity labels to secure distributed executions. Its goal—combining different cryptographic primitives including MPC, zero-knowledge proofs, and commitment schemes—is orthogonal to ours and its scope is broader. It does not consider cost-based hybrid protocol mixing for MPC.

Other cryptographic compilers. Further afield, there is much work on compilers for other cryptographic primitives. Compilers for fully homomorphic encryption (FHE) [20], [67], [68], [66], [9], [3], [69], [25], [24], [17], [15], [21] support computation over encrypted data, enabling privay-preserving outsourcing. Compared to MPC, FHE requires less interaction, at the expense of compute efficiency. Compilers for zero-knowledge proofs (ZKPs) [64], [63], [11], [70], [56], [19], [42], [53], [6], [27], [75], [18] support privacy-preserving proofs about secret data. ZKPs can achieve public verifiability, but apply only when all secret information is known by one party (the prover); this substantially alters the compilation problem. Recent works [62], [5], [39], [54] blur the line between ZKPs and MPC using techniques orthogonal to ours.

10. Conclusion

Silph is a scalable, efficient, and accurate hybrid protocol compiler for MPC programs written in high-level languages. Our evaluation shows that Silph greatly improves the overall hybrid MPC compilation process and can deliver optimized, fine-grained protocol assignments across various database analytics and machine learning workloads.

11. Acknowledgements

We'd like to give many thanks to the anonymous S&P reviewers and our shepherd for their timely and detailed feedback. We'd also like to thank Bernardo Subercaseaux for his helpful discussions and insights in formulating our inter-partition ILP. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1745016, and funding from PNC, Samsung, and CyLab Seed funding. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] ACAY, C., RECTO, R., GANCHER, J., MYERS, A. C., AND SHI, E. Viaduct: an extensible, optimizing compiler for secure distributed programs. In *PLDI* (2021).

[2] ALY, A., CONG, K., COZZO, D., KELLER, M., ORSINI, E., ROTARU, D., SCHERER, O., SCHOLL, P., SMART, N., TANGUY, T., ET AL. SCALE-MAMBA. https://homes.esat.kuleuven.be/~nsmart/SCALE/.

[3] ARCHER, D. W., CALDERÓN TRILLA, J. M., DAGIT, J., MALOZEMOFF, A., POLYAKOV, Y., ROHLOFF, K., AND RYAN, G. Ramparts: A programmer-friendly system for building homomorphic encryption applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (2019).

[4] BARRETT, C., STUMP, A., AND TINELLI, C. The SMT-LIB standard: Version 2.0. In *SMT* (2010).

[5] BAUM, C., DAMGÅRD, I., AND ORLANDI, C. Publicly auditable secure multi-party computation. In *SCN* (2014).

[6] BAYLINA, J. Circom. https://github.com/iden3/circom.

[7] BEAVER, D., MICALI, S., AND ROGAWAY, P. The round complexity of secure protocols. In *STOC* (1990).

[8] BEN-OR, M., GOLDWASSER, S., AND WIGDERSON, A. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC* (1988).

[9] BOEMER, F., LAO, Y., CAMMAROTA, R., AND WIERZYNSKI, C. ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers* (2019).

[10] BONAWITZ, K., IVANOV, V., KREUTER, B., MARCEDONE, A., MCMAHAN, H. B., PATEL, S., RAMAGE, D., SEGAL, A., AND SETH, K. Practical secure aggregation for privacy-preserving machine learning. In *CCS* (2017).

[11] BRAUN, B., FELDMAN, A. J., REN, Z., SETTY, S., BLUMBERG, A. J., AND WALFISH, M. Verifying computations with state. In *SOSP* (2013). Extended version: http://eprint.iacr.org/2013/356.

[12] BRAUN, L., DEMMLER, D., SCHNEIDER, T., AND TKACHENKO, O. Motion–a framework for mixed-protocol multi-party computation. *ACM Transactions on Privacy and Security 25*, 2 (2022), 1–35.

[13] BÜSCHER, N., DEMMLER, D., KATZENBEISSER, S., KRETZMER, D., AND SCHNEIDER, T. Hycc: Compilation of hybrid protocols for practical secure computation. In *CCS* (2018).

[14] California Consumer Privacy Act (CCPA) 2018. https://oag.ca.gov/privacy/ccpa, 2018.

[15] CARPOV, S., DUBRULLE, P., AND SIRDEY, R. Armadillo: a compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing* (2015).

[16] CHANDRAN, N., GUPTA, D., RASTOGI, A., SHARMA, R., AND TRIPATHI, S. Ezpc: programmable and efficient secure two-party computation for machine learning. In *Euro S&P* (2019).

[17] CHIELLE, E., MAZONKA, O., GAMIL, H., TSOUTSOS, N. G., AND MANIATAKOS, M. E3: A framework for compiling c++ programs with encrypted operands. *Cryptology ePrint Archive* (2018).

[18] CHIN, C., WU, H., CHU, R., COGLIO, A., MCCARTHY, E., AND SMITH, E. Leo: A programming language for formally verified, zero-knowledge applications, 2021. https://ia.cr/2021/651.

[19] COSTELLO, C., FOURNET, C., HOWELL, J., KOHLWEISS, M., KREUTER, B., NAEHRIG, M., PARNO, B., AND ZAHUR, S. Geppetto: Versatile verifiable computation. In *IEEE S&P* (2015).

[20] COWAN, M., DANGWAL, D., ALAGHI, A., TRIPPEL, C., LEE, V. T., AND REAGEN, B. Porcupine: A synthesizing compiler for vectorized homomorphic encryption. In *PLDI* (2021).

[21] CROCKETT, E., PEIKERT, C., AND SHARP, C. Alchemy: A language and compiler for homomorphic encryption made easy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018).

[22] DAMGÅRD, I., PASTRO, V., SMART, N., AND ZAKARIAS, S. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO* (2012).

[23] DAMGÅRD, I., PASTRO, V., SMART, N., AND ZAKARIAS, S. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO* (2012).

[24] DATHATHRI, R., KOSTOVA, B., SAARIKIVI, O., DAI, W., LAINE, K., AND MUSUVATHI, M. Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2020).

[25] DATHATHRI, R., SAARIKIVI, O., CHEN, H., LAINE, K., LAUTER, K., MALEKI, S., MUSUVATHI, M., AND MYTKOWICZ, T. Chet: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019).

[26] DEMMLER, D., SCHNEIDER, T., AND ZOHNER, M. Aby-a framework for efficient mixed-protocol secure two-party computation. In *NDSS* (2015).

[27] EBERHARDT, J., AND TAI, S. Zokrates - scalable privacy-preserving off-chain computations. In *IEEE Blockchain* (2018).

[28] FANG, V., BROWN, L., LIN, W., ZHENG, W., PANDA, A., AND POPA, R. A. Costco: An automatic cost modeling framework for secure multi-party computation. *Cryptology ePrint Archive* (2022).

[29] FORREST, J., AND LOUGEE-HEIMER, R. Cbc user guide. In *Emerging theory, methods, and applications*. INFORMS, 2005, pp. 257–277.

[30] GDPR. Official Journal of the European Union '16.

[31] GILAD-BACHRACH, R., DOWLIN, N., LAINE, K., LAUTER, K., NAEHRIG, M., AND WERNSING, J. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *ICML* (2016).

[32] HAZAY, C., AND VENKITASUBRAMANIAM, M. Scalable multi-party private set-intersection. In *IACR International Workshop on Public Key Cryptography* (2017).

[33] HOLZER, A., FRANZ, M., KATZENBEISSER, S., AND VEITH, H. Secure two-party computations in ANSI C. In *CCS* (2012).

[34] HUANG, Y., EVANS, D., KATZ, J., AND MALKA, L. Faster secure two-party computation using garbled circuits. In *USENIX Security* (2011).

[35] ISHAQ, M., MILANOVA, A. L., AND ZIKAS, V. Efficient mpc via program analysis: A framework for efficient optimal mixing. In *CCS* (2019).

[36] JUVEKAR, C., VAIKUNTANATHAN, V., AND CHANDRAKASAN, A. Gazelle: A low latency framework for secure neural network inference. In *USENIX Security* (2018).

[37] Kahip - karlsruhe high quality partitioning. https://kahip.github.io/.

[38] Kahypar - karlsruhe hypergraph partitioning. https://github.com/kahypar/kahypar.

[39] KANJALKAR, S., ZHANG, Y., GANDLUR, S., AND MILLER, A. Publicly auditable mpc-as-a-service with succinct verification and universal setup. In *Euro S&P* (2021).

[40] KELLER, M., ORSINI, E., AND SCHOLL, P. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *CCS* (2016).

[41] KELLER, M., PASTRO, V., AND ROTARU, D. Overdrive: making spdz great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2018).

[42] KOSBA, A. E., PAPAMANTHOU, C., AND SHI, E. xJsnark: A framework for efficient verifiable computation. In *IEEE S&P* (2018).

[43] LEHMKUHL, R., MISHRA, P., SRINIVASAN, A., AND POPA, R. A. Muse: Secure inference resilient to malicious clients. In *USENIX Security* (2021).

[44] LINDELL, Y., PINKAS, B., SMART, N. P., AND YANAI, A. Efficient constant-round multi-party computation combining bmr and spdz. *Journal of Cryptology* (2019).

[45] LIU, C., HUANG, Y., SHI, E., KATZ, J., AND HICKS, M. Automating efficient ram-model secure computation. In *IEEE S&P* (2014).

[46] LIU, C., WANG, X. S., NAYAK, K., HUANG, Y., AND SHI, E. Oblivm: A programming framework for secure computation. In *IEEE S&P* (2015).

[47] LIU, J., JUUTI, M., LU, Y., AND ASOKAN, N. Oblivious neural network predictions via minionn transformations. In *CCS* (2017).

[48] MALKHI, D., NISAN, N., PINKAS, B., SELLA, Y., ET AL. Fairplay-secure two-party computation system. In *USENIX Security* (2004).

[49] MISHRA, P., LEHMKUHL, R., SRINIVASAN, A., ZHENG, W., AND POPA, R. A. Delphi: A cryptographic inference service for neural networks. In *USENIX Security* (2020).

[50] MOHASSEL, P., AND ZHANG, Y. Secureml: A system for scalable privacy-preserving machine learning. In *IEEE S&P* (2017).

[51] NIELSEN, J. B., NORDHOLT, P. S., ORLANDI, C., AND BURRA, S. S. A new approach to practical active-secure two-party computation. In *CRYPTO* (2012).

[52] NIKOLAENKO, V., WEINSBERG, U., IOANNIDIS, S., JOYE, M., BONEH, D., AND TAFT, N. Privacy-preserving ridge regression on hundreds of millions of records. In *IEEE S&P* (2013).

[53] Noir. https://noir-lang.github.io/book/index.html.

[54] OZDEMIR, A., AND BONEH, D. Experimenting with collaborative zk-snarks: Zero-knowledge proofs for distributed secrets. In *USENIX Security* (2022).

[55] OZDEMIR, A., BROWN, F., AND WAHBY, R. S. Circ: Compiler infrastructure for proof systems, software verification, and more. In *IEEE S&P* (2022).

[56] PARNO, B., HOWELL, J., GENTRY, C., AND RAYKOVA, M. Pinocchio: Nearly practical verifiable computation. In *IEEE S&P* (2013).

[57] PATRA, A., SCHNEIDER, T., SURESH, A., AND YALAME, H. {ABY2. 0}: Improved {Mixed-Protocol} secure {Two-Party} computation. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 2165–2182.

[58] RASTOGI, A., HAMMER, M. A., AND HICKS, M. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *IEEE S&P* (2014).

[59] RIAZI, M. S., WEINERT, C., TKACHENKO, O., SONGHORI, E. M., SCHNEIDER, T., AND KOUSHANFAR, F. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security* (2018).

[60] SANDERS, P., AND SCHULZ, C. Think locally, act globally: Highly balanced graph partitioning. In *SEA* (2013).

[61] SCHLAG, S. *High-Quality Hypergraph Partitioning*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2020.

[62] SCHOENMAKERS, B., VEENINGEN, M., AND VREEDE, N. D. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In *ANCS* (2016).

[63] SETTY, S., BRAUN, B., VU, V., BLUMBERG, A. J., PARNO, B., AND WALFISH, M. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys* (2013).

[64] SETTY, S. T. V., VU, V., PANPALIA, N., BRAUN, B., BLUMBERG, A. J., AND WALFISH, M. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security* (2012). Extended version: https://ia.cr/2012/598.

[65] TORLAK, E., AND BODIK, R. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI* (2014).

[66] VAN ELSLOO, T., PATRINI, G., AND IVEY-LAW, H. Sealion: A framework for neural network inference on encrypted data. *arXiv preprint arXiv:1904.12840* (2019).

[67] VIAND, A., JATTKE, P., HALLER, M., AND HITHNAWI, A. HECO: Automatic code optimizations for efficient fully homomorphic encryption. *arXiv preprint arXiv:2202.01649* (2022).

[68] VIAND, A., JATTKE, P., AND HITHNAWI, A. Sok: Fully homomorphic encryption compilers. In *2021 IEEE Symposium on Security and Privacy (SP)* (2021).

[69] VIAND, A., AND SHAFAGH, H. Marble: Making fully homomorphic encryption accessible to all. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (2018).

[70] WAHBY, R. S., SETTY, S., REN, Z., BLUMBERG, A. J., AND WALFISH, M. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS* (2015).

[71] WANG, X., RANELLUCCI, S., AND KATZ, J. Global-scale secure multiparty computation. In *CCS* (2017).

[72] YAO, A. C.-C. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science* (1986), SFCS '86.

[73] ZAHUR, S., AND EVANS, D. Obliv-C: A language for extensible data-oblivious computation. https://ia.cr/2015/1153, 2015.

[74] ZHENG, W., DENG, R., CHEN, W., POPA, R. A., PANDA, A., AND STOICA, I. Cerebro: A platform for {Multi-Party} cryptographic collaborative learning. In *USENIX Security* (2021).

[75] Zinc. https://zinc.matterlabs.dev/.