Efficient MPC via Program Analysis: A Framework for Efficient Optimal Mixing[*]

Muhammad Ishaq[1][**], Ana L. Milanova[2][***], and Vassilis Zikas[1][†]

[1] University of Edinburgh, Edinburgh, Scotland
{m.ishaq, vassilis.zikas}@inf.ed.ac.uk
[2] Rensselaer Polytechnic Institute (RPI), Troy, New York
milanova@cs.rpi.edu

Abstract Multi-party computation (MPC) protocols have been extensively optimized in an effort to bring this technology to practice, which has already started bearing fruits. The choice of which MPC protocol to use depends on the computation we are trying to perform. Protocol mixing is an effective black-box —with respect to the MPC protocols—approach to optimize performance. Despite, however, considerable progress in the recent years existing works are heuristic and either give no guarantee or require an exponential (brute-force) search to find the optimal assignment, a problem which was conjectured to be NP hard.
We provide a theoretically founded approach to optimal (MPC) protocol assignment, i.e., optimal mixing, and prove that under mild and natural assumptions, the problem is tractable both in theory and in practice for computing best two-out-of-three combinations. Concretely, for the case of two protocols, we utilize program analysis techniques—which we tailor to MPC—to define a new integer program, which we term the *Optimal Protocol Assignment* (in short, OPA) problem whose solution is the optimal (mixed) protocol assignment for these two protocols. Most importantly, we prove that the solution to the linear program corresponding to the relaxation of OPA is integral, and hence is also a solution to OPA. Since linear programming can be efficiently solved, this yields the first efficient protocol mixer. We showcase the quality of our OPA solver by applying it to standard benchmarks from the mixing literature. Our OPA solver can be applied on any two-out-of-three protocol combinations to obtain a best two-out-of-three protocol assignment.

Keywords: protocol mixing, linear programming, multiparty computation, program analysis, cryptography

1 Introduction

Multi-party computation (in short, MPC) allows M parties $p_1, \ldots, p_M$ to perform any given computation on their private inputs in a secure manner. Informally, security means that the protocol should correctly compute the specified output (correctness) and it should not leak any information about the inputs, other than what can be deduced from this output (privacy).

From the onset of MPC [Yao82; GMW87; BGW88; CCD88], there have been two approaches to MPC protocol design: (1) the so-called *garbled-circuit*-based approach, also referred to as Yao's protocol [Yao82], and (2) the approach following the secret-sharing-based (aka gate-by-gate evaluation) paradigm. The latter was introduced by Goldreich, Micali and Wigderson (and is therefore often referred to as the GMW approach) [GMW87]; GMW works with boolean circuits and was extended by Ben-Or, Goldwasser, and Wigderson in [BGW88]—the so-called BGW protocol—to compute arithmetic circuits over finite fields.

The above approaches have inherent quantitative differences. First, the round complexity of gate-by-gate evaluation is linear to the (multiplicative) depth of the circuit, whereas Yao's approach yields constant-round

[*] This is full version of the work[IMZ19] that appeared in ACM CCS'19. (Last Update: July 12, 2019)
[**] This work was done in part while the author was at RPI.
[***] This work was done while the author was supported in part by NSF awards 1319384 and 1814898.
[†] This work was done in part while the author was visiting UCLA and supported in part by DARPA and SPAWAR under contract N66001-15-C-4065 and by a SICSA Cyber Nexus Research Exchanges grant.

protocols; furthermore, unlike Yao's protocol most of the costly computation and communication in GMW can be outsourced to an offline (pre-computation) phase which is independent of the inputs. Thus, GMW (assuming preprocessing) is often more efficient over a wide area network (WAN) where communication can be the bottleneck[3] and garbled-citcuit-based approaches, which inherently needs to communicate a lot of information, is faster assuming fast networks.

Furthermore, all the above protocols have communication and/or computation proportional to the size of the circuit they aim to compute. For example, as demonstrated in [DSZ15], it is much faster to use garbled-circuits on the standard Boolean circuit for comparison, than using GMW on a state-of-the-art representations of comparison as an arithmetic circuit. On the other hand, performing multiplication (of bounded precisions floats or bounded size integers) is much faster by means of GMW for an appropriately large field—it is effectively computing a single-gate circuit—than by means of Yao's protocol applied on the state-of-the-art Boolean circuit for field multiplication.

The above demonstrates that there is no "one size fit all" solution to optimal MPC. In order to decide what protocol to use, one would need to take into account both the target computation, and the parameters of the network. This might be feasible for simple computations, e.g., only comparisons or only multiplications, but it becomes challenging when we are aiming to perform a complicated computation, whose circuit is not even a-priori known. To cope with this, the idea of mixed (or hybrid) protocols has been proposed [KSS13; Hen+10; BLW08; SKM11; Cho+13]. These are protocols that evaluate different parts of the computation by means of different protocols, e.g., part of the computation is performed using garbled circuits, and another part is performed using GMW.

Deciding which part of the computation should be computed using which protocol is a challenging task. One of the reasons is that one needs to come up with an appropriate cost model, that estimates the costs for computing each part of the specification with each of the candidates. Such costs were recently calculated in [DSZ15] for hybrid protocols combining a garbled-circuits-based protocol, with two versions of GMW, one for computing arithmetic circuits over arithmetic fields of characteristic 2 and size k, i.e., $\mathbb{Z}_2^k$, and one for computing Boolean circuits (i.e., arithmetic circuits over $\mathbb{Z}_2$). Concretely, they devised benchmarks that estimate for different useful computations, which protocol is fastest in different scenarios.

One would hope that such a cost allocation would already reduce the problem to a simple optimization problem: try to split the computation in modules from a predefined set, and then compute the optimal allocation of protocols depending on the cost of each module. Unfortunately this intuition is overly simplistic as discussed below.

First, one needs to take into account the need to stitch different modules together in a way that does not reveal information. In order to do so the protocol needs to allow each sub-protocol to pass its (output) state to the next sub-protocol. This can be done by computing and outputting a secret sharing of the state that is then given as input to the next module. Albeit, different protocols handle different types of sharing, e.g., in GMW the shares are field elements, whereas in Yao's protocol one needs the inputs to be Boolean. This means that in order for a GMW computed module to pass state to a Yao module it needs to convert its sharing to a Boolean sharing. Such a conversion would typically involve (secure) bit-decomposition of $\mathbb{Z}_2^k$ elements which is an expensive operation. Hence, in order to decide whether it is worth switching from GMW to garbled-circuits, one needs to take into account the cost of converting the associated shares.

Second, such conversion costs need to be incorporated in addition to the cost of module computation. A model incorporating such costs into an optimization problem was introduced in [SK11; KSS14], where the authors also specified an Integer Program (IP) computing the optimal solution. Due to the difficulty of solving Integer Programming in general, this lead to a conjecture that the problem of optimal protocol assignment is NP-hard. The conjecture was adopted by follow-up works [Pat+16; Cha+17; Büs+18] and gave rise to heuristic approaches.

[3] This is demonstrated in existing benchmarks [DSZ15; Büs+18] (including ours) which are run for the semi-honest setting and do not account for the cost of synchrony, e.g., timeouts, hence the effect of the increased round complexity in GMW is minimized.

1.1 Our Results

In this work we show that the problem of optimal (MPC) protocol mixing is tractable (efficiently solvable) for for the case of combinations of two multi-party protocols. In a nutshell, starting from non-annotated source-code, we employ a combination of program analysis and combinatorial optimization techniques to devise an integer program, which we term the *optimal protocol-assignment* problem (OPA). OPA yields a provably optimal mixing—up to parallelisation/scheduling and compiler optimizations (see §4 for details)—and, as we prove, accepts a polynomial-time solver.

We remark that our current approach does not directly extend to the case of three protocols (see Remark 3 for details). Thus the question of whether or not OPA can be efficiently solved in the three-protocol case remains open. However, since the optimal two-protocol combination can be found in polynomial time, we can use our solver to compute best two-out-of-c protocols combinations for a constant c, by applying it to all possibles pairs of the c protocols, and picking the pair (and the corresponding solution) that minimizes the objective function across all $\binom{c}{2}$ applications.

To demonstrate the quality of our OPA solver, we apply it to compute protocol assignments for known benchmarks from ABY [DSZ15] and HyCC [Büs+18], for which code has been released namely Modular Exponentiation, Biometric Matching and Private Set Intersection, Convolutional Neural Networks (CNNs) of MiniONN [Liu+17] and Crytonets [Gil+16], k-means clustering algorithm, DB Merge and DBJoin as well as new ones we introduce, namely Greatest Common Divisor (GCD) and Histogram. We remark that works [DSZ15; Büs+18] directly compute optimal assignment for three protocols (this is done by manual assignment in [DSZ15], and exhaustive search in [Büs+18]). However, with the exception of Modular Exponentiation in the LAN setting, all resulting optimal assignments use one protocol or a mixing of only two protocols. This state of practice indicates that our solver can be used to compute optimal assignments for three protocols.

More concretely, our contributions can be summarized as follows:

As our main result, we prove that the Optimal Protocol Assignment (OPA) problem for two protocols is, in fact, tractable. To this direction, we put forth a framework combining methods from program analysis and MPC, and establishing a common language between the two disciplines. This framework allows us to formally specify the OPA and describe all the relevant parameters of an integer program (IP), such that given MPC code, it computes the optimal assignment of the two given protocols (and their share conversions). We use our model to show that the linear-programming (LP) relaxation of our IP has an integral solution, and therefore OPA is polynomial-time solvable.[4] In addition to offering the language for stating and proving our results, we set forth problems for the programming languages/compilers community, that can lead to improvement in MPC compilers.

The running time of our OPA solver is polynomial on the size of its input (i.e., the MPC code). To provide a more practical implementation, we propose *MPC-source* as an abstraction of MPC code. MPC-source is a representation of the original MPC code which enables static analysis, while it is substantially more compact than standard linearized MPC code—i.e., the straight-line version of MPC code. In particular, MPC source has significantly fewer variables and statements than linearized MPC, thereby reducing the search space for the optimal protocol combination. We show how to apply OPA on MPC-source, and prove that under natural assumptions on the optimal assignment computed by $IP_{\mathsf{Linear}(S)}$, an OPA solutions for MPC-source is optimal for the linearized MPC. We note in passing that although making the treatment more involved, devising such a faster and more scalable solver is useful for deriving a practical solution to the problem. Notwithstanding, our entire treatment can be applied on linearized MPC code as well.

Finally, to demonstrate the practicality of our solver, we provide a toolchain that takes high-level unannotated source—Java source code in our case—, translates it to MPC-source, and outputs optimal protocol assignments. We compare our solver with publicly available benchmarks from [DSZ15; Büs+18]. We remark that the concrete assignment from [DSZ15; Büs+18] is known for only a subset of the related benchmarks; for all those we confirm (for 2-out-of-3 protocols) the same assignment. For the remainder, we provide the

[4] Unlike integer programming which is known to be NP-hard, linear programming is solvable in polynomial time [Kar84; Kha80].

first publicly released assignments, and compare our resulting protocol combinations with the ones reported in [DSZ15; Büs+18]. Our solver is available on GitHub (`https://github.com/ishaq/OPA`).

We believe that our work opens possibilities for future work. Program synthesis, program analysis of MPC-source, e.g., program equivalence and parallelization, as well as integration of OPA into MPC compilers, are some of the possibilities. Throughout this paper, we pose conjectures and outline future directions.

1.2 Comparison to Related Work

A number of works have demonstrated the advantages of mixed protocols [KSS13; Hen+10; BLW08; SKM11; Cho+13]. The ABY framework [DSZ15] by Demmler et.al. provided easy to use framework for writing mixed-protocol 2-party computations. Mohassel and Rindal [MR18] improved it into ABY^3 and extended it to 3 parties. However, all these works require the programmer to manually choose protocol assignment. In contrast, our tool yields automatic solution to optimal protocol assignment in polynomial time. We note, however, that previous works compute an optimal assignment among all three protocols whereas our tool can, so far, handle only two protocols. In fact, our analysis does not directly extend to three protocols. Thus, the question of whether or not OPA for three or more protocol is NP-hard remains open.

Yet, by using our tool three times, once with each pair of the three protocols from [DSZ15], and keeping the overall optimal solution we obtain a tool for finding, in polynomial time, the best two-out-of-three protocol combination. Interestingly, for the overwhelming majority of existing benchmarks this extension yields assignments consistent to the original exhaustive search method—the reason is that the existing assignment among three protocols for these benchmarks ends up using at most two of them. This allows us to handle arbitrary long code for which exhaustive search might be infeasible.

The work by Kerschbaum et.al. [KSS14] was the first to discuss the problem of automatic protocol selection. They require the source program to be expressed in straight-line three-address representation and formulate a 0-1 integer program for the two-protocol case. The integer program computes the optimal assignment. Since 0-1 integer programming is NP-hard, this lead to the conjecture that the optimal protocol assignment problem is NP hard. In fact, Kerschbaum et.al. [KSS14] proposes the first heuristics for solving the above problem. We note that despite some similarities, e.g., some common inequalities, of the IP from [KSS14] to the one underlying our OPA problem, we were unable to find a way to prove that the LP relaxation of their IP has an integral solution. Instead, here we provide our new IP which leverages our model to allow us to prove existence of an efficient solver.

EzPC [Cha+17] is a recent work that takes a high-level imperative language as input and compiles it to mixed-protocols ABY source code. It is also based on heuristics. Moreover its heuristics do not take into account dependencies between different parts of the code (i.e., they only rely on local information) and are, therefore, too weak. For example, they state that their compiler never compiles a multiplication into a Boolean/Yao representation. On a high bandwidth network with low latency (typical case of 10Gbps LAN), it is actually inefficient to do so if the number of multiplications is small and un-amortized.

Most recently, a mixed protocol compiler, called HyCC [Büs+18], was introduced that uses a combination of exhaustive search and heuristics to optimize and automate mixing. The unit of optimization in HyCC is a module, which can be as little as one instruction but the sheer number of choices for exhaustive search or heuristics make it prohibitive to have such fine granularity. In contrast, we provide provably optimal mixing conditioned on a fixed schedule and access to the SSA-representation of the input program.

2 Preliminaries

We review the basic notions from the related MPC literature and establish some necessary terminology and notation. Our work combines and extends techniques from cryptography, in particular MPC, with program analysis, and combinatorial optimization. Since this might require a combination of expertise, in Appendix B we review basic program analysis concepts that are useful for evaluating our results.

We will consider the optimal protocol assignment (OPA) problem for deriving hybrid (i.e., mixed) protocols against *semi-honest,* aka passive, adversaries—who follow their protocol instructions but attempt to acquire more information than the specified output by analyzing their (joint) view of the computation. We note in passing that although, consistent with existing literature, our experiments are for semi-honest two-party protocols only, our theory, and in particular our feasibility result for solving OPA, directly applies to malicious and or multi-party protocols.

In our experiments we focus on protocols that combine the same three types of semi-honest MPC protocols as in [DSZ15] as it will allow us to use the primitive-MPC cost estimators introduced there. In the following we give the high level description of these protocols and the associated sharing, and refer to [DSZ15] for a detailed description of the optimization thereof. We stress that our program analysis technique can be applied to any version of these protocols (with or without such optimizations.)

Secret Sharing A t-out-of-n secret sharing scheme allows a dealer (or a protocol) to share a value s among n parties, such that the shares of any $t-1$ parties leak no information on s, but the shares of any t parties uniquely define s. In this work we focus on two-party computation—although our theory applies to the three-party case along the lines of [MR18]. More concretely, a value is shared among the two parties $\{p_1, p_2\}$ if every party p_i holds a *share* $\langle s\rangle_i$ such that there exist a reconstruction algorithm which given both $\langle s\rangle_1$ and $\langle s\rangle_2$ outputs s, but each $\langle s\rangle_i$ by itself contains no information on s. We will denote the vector of shares by $\langle s\rangle = (\langle s\rangle_1, \langle s\rangle_2)$ and refer to it as a *sharing of* s.

The MPC modules: The three (types of) MPC protocols, also referred to as MPC modules, that will be considered here (and their associated secret sharing schemes) are as follows (cf. [DSZ15] for more details on the specific optimizations):

- A: $\pi^{\mathtt{A}}$ is a protocol for computing arithmetic circuits over the finite field $\mathbb{Z}_{2^k}$. Such a protocol uses the BGW gate-by-gate evaluation paradigm, where so-called Beaver multiplication triples [Bea92]—which can be pre-computed—are used to make the online phase linear.[5] Concretely, the protocol stores each value s in its state as an *arithmetic secret sharing*, denoted by $\langle s\rangle^{\mathtt{A}}$: Each p_i holds a share $\langle s\rangle_i \in \mathbb{Z}_{2^k}$ such that $\langle s\rangle_1^{\mathtt{A}} + \langle s\rangle_2^{\mathtt{A}} \equiv s \pmod{2^\ell}$. (Consistently with [DSZ15], for clarity we will denote the type of the sharing by a letter A the exponent.) As demonstrated in [DSZ15], with the appropriate optimizations $\pi^{\mathtt{A}}$ is the best known protocol for arithmetic operations, primarily in WAN setting but also in LAN setting if sufficiently amortized.
- B: $\pi^{\mathtt{B}}$ is a protocol for computing Boolean circuits based on GMW. It uses the XOR sharing which is the same as the arithmetic sharing but for $\mathbb{Z}_2$, i.e., a bit s is shared by bits $\langle s\rangle_1^{\mathtt{B}}$ and $\langle s\rangle_2^{\mathtt{B}}$, s.t., $\langle s\rangle_1^{\mathtt{B}} \oplus \langle s\rangle_2^{\mathtt{B}} = s$. As demonstrated in [DSZ15], with the appropriate optimizations $\pi^{\mathtt{B}}$ is the best known protocol for comparisons and logical operations in LAN setting, provided the operations are amortized.
- Y: Finally, we will denote by $\pi^{\mathtt{Y}}$ the (optimized) version of Yao's protocol used in [DSZ15]. For brevity, we refer to $\pi^{\mathtt{Y}}$ as the *Yao-based protocol.* Note that although the original Yao protocol does not operate on secret shared value, one can interpret the state, i.e., for each wire of the Boolean circuit, the corresponding value s_w of the wire w, as being shared among the to parties as follows: P_1, the circuit creator, holds the two keys K_0^w and K_1^w corresponding to wire inputs 0 and 1 respectively, and P_2 holding $K_{s_w}^w$, i.e., $\langle s_w\rangle_1^{\mathtt{Y}} = (K_0^w, K_1^w)$ and $\langle s_w\rangle_2^{\mathtt{Y}} = K_{s_w}^w$. Clearly, in $\langle s_w\rangle^{\mathtt{Y}} = (\langle s_w\rangle_1^{\mathtt{Y}}, \langle s_w\rangle_2^{\mathtt{Y}})$, p_1 does not know s_w and p_2 does not known which value $K_{s_w}^w$ corresponds to. Hence, none of the parties knows s_w but by pooling their shares together they can easily reconstruct by checking if $K_{s_w}^w$ equals K_1^w. We refer to this secret sharing scheme as *Yao sharing.* As demonstrated in [DSZ15], with the appropriate optimizations $\pi^{\mathtt{Y}}$ is the best known protocol for comparisons and logical operations, especially in LAN setting.

Share conversion As discussed above, in order to stitch different modules in a single protocol we need to transform the (output) sharing of one module to the (input) sharing of the following module. There are several such share conversion protocols. In our benchmarks we use the ones from [DSZ15] but our OPA solver can be instantiated with any such protocol. We refer to the share conversion protocol that converts sharing of type X to sharing of type Y as `X2Y`, where `X` and `Y` take the value A for arithmetic, B for Boolean, and Y for Yao sharing. E.g., a share conversion protocol from arithmetic to Yao sharing is denoted by `A2Y`.

[5] Looking ahead, the costs used in our empirical study will be be sum of the setup and online costs.

```
int gcd(int a, int b) {
  int x = a;
  int y = b;
  for (int i = 0; i < 2*LEN; i++)
  {
    if (y != 0)
    {
      int r = rem(x,y);
      x = y;
      y = r;
    }
  }
  return x;
}

// returns val%mod
int rem(int val, int mod) {
  int rem = 0;
  for (int j = LEN-1; j ≥ 0; j--)
  {
    rem = rem << 1;
    // rem[0] = val[j]
    rem = rem + ((val>>j)&1);
    if (rem ≥ mod)
    {
      rem = rem - mod;
    }
  }
  return rem;
}
```

(a) IMP Source

```
int gcd(int a, int b) {
  int x0 = a;
  int y0 = b;
  for (int i = 0; i < 2*LEN; i++)
  {
    x1 = (i == 0) ? x0 : x3;
    y1 = (i == 0) ? y0 : y3;

    if (y1 != 0)
    {
      // begin inlined rem
      int rem0 = 0;
      for (int j = LEN-1; j ≥ 0; j--)
      {
        rem1 = (j==LEN-1) ? rem0 : rem5;
        rem2 = rem1 << 1;
        rem3 = rem2 + (x1>>j)&1;
        if (rem3 ≥ y1)
        {
          rem4 = rem3 - y1;
        }
        rem5 = φ(rem4,rem3);
      }
      // end inline rem
      int r = rem5;
      x2 = y1;
      y2 = r;
    }
    x3 = φ(x2,x1);
    y3 = φ(y2,y1);
  }
  return x3;
}
```

(b) IMP-SSA

```
int gcd(int a, int b) {
  int x0 = a;
  int y0 = b;
  for(int i = 0; i < 2*LEN; i++) {
    x1 = (i == 0) ? x0 : x3;
    y1 = (i == 0) ? y0 : y3;

    // begin inlined rem
    int rem0 = 0;
    for (int j = LEN-1; j >= 0; j--)
    {
      rem1 = (j==LEN-1) ? rem0 : rem5;
      rem2 = rem1 << 1;
      rem3 = rem2 + (x1>>j)&1;
      rem4 = rem3 - y1;
      cnd1 = CMP(rem3 >= y1);
      rem5 = MUX(rem3,rem4,cnd1);
    }
    // end inline rem

    int r = rem5;
    x2 = y1;
    y2 = r;
    cnd2 = CMP(y1 != 0);
    x3 = MUX(x1,x2,cnd2);
    y3 = MUX(y1,y2,cnd2);
  }
  return x3;
}
```

(c) MPC-source

Figure 1. (a) shows the IMP source for the GCD algorithm, (b) shows GCD translated into IMP-SSA after inlining rem. (c) shows the IMP-SSA program translated into MPC-source. Our integer program works on MPC-source.

3 Program Analysis of MPC Source

In this section, we describe our program analysis process, that will yield the basis for our optimization problem defined in the next section. §3.1 presents a running example. §3.2 outlines the syntax of the source language, as well as the translation process into our representation, MPC-source. §3.3, and §3.4 describe the control-flow structure of MPC-source and reaching definition analysis on top of it. §3.5 and §3.6 define other analyses on MPC-source necessary to build the optimization problem.

3.1 Running Example

Our running example in Fig. 1(a) is an implementation of the Greatest Common Divisor (GCD) algorithm using integer division. The gcd program makes calls to function rem, due to [DSZ15], which computes the remainder of an integer division. Note that the structure is significantly different and more involved than the standard—non-MPC targeted—integer divisiongiven in Fig. 7(a) in appendix C. Such difference between non-MPC and MPC programs is typical due to inherent restrictions in the latter (to preserve privacy). For example, in Fig. 1(a) the value of both val and mod will need to be secret shared, so they remain unknown until the corresponding output-gates of the induced MPC circuit are computed (and reconstructed). Thus, in order to generate a circuit that can be processed by MPC, the while-loop cannot use the values of these variables. The rewrite by Demler et al. [DSZ15] rectifies this by carrying long division in binary, with a loop bounded by statically known LEN, which is either 32 or 64 bits. Fig. 1(a) presents our rewrite of the standard GCD loop (cf. Fig. 7(b) in appendix C), where we are using the observation that the number of iterations in GCD is bounded by $2\mathsf{LEN} = 2\log(\max(a, b))$.

$s ::= s_1; s_2 \Rightarrow s.MPC = s_1.MPC + s_2.MPC$
$s ::= \mathsf{if}\ (\mathsf{x\ bop\ y})\ \{\ s_1\ \}\ \mathsf{else}\ \{\ s_2\ \}\ \mathsf{z} = \phi(\mathsf{z_1}, \mathsf{z_2}) \Rightarrow s.MPC = s_1.MPC + s_2.MPC +$ "$\mathsf{cnd} = \mathsf{CMP}(\mathsf{x\ bop\ y}); \mathsf{z} = \mathsf{MUX}(\mathsf{z_1}, \mathsf{z_2}, \mathsf{cnd})$"

Figure 2. Translation of IMP-SSA into MPC-source. Attribute *MPC* contains the MPC-source code. Translation of a sequence entails appending s_2's MPC-source code onto s_1's. The MPC-source for an if-statement is constructed by adding the code for branch s_2 onto the code for branch s_1 thus linearizing the if-statement; at the end, the translation adds the conditional operation and the multiplexer, which selects values. We do not include for other kinds of statements as it is trivial.

3.2 Translation into MPC-source

We assume an IMP-like syntax [NK14] for our source language. The IMP syntax models an imperative language, such as FORTRAN, C, or Java, and our results apply to any of these languages. We impose the following standard restrictions necessary to accommodate MPC: there is no recursion, and all loop bounds are statically known. The IMP source is translated into Static Single Assignment (SSA) using standard techniques [Cyt+91]. This is standard SSA, however, to make it explicit that it corresponds to IMP-source, in the following we will refer to it as *IMP-SSA*. This is the syntax of our intermediate representation. Due to space constraints, we defer detailed discussion of the syntax to the Appendix C.

The next step is to translate IMP-SSA into *MPC-source*, the representation that we use for defining our compact integer program. Fig. **??** defines an *attribute grammar* (also known as *syntax-directed translation*) over IMP-SSA. The most interesting case arises at if-statements which are dealt with using standard MPC techniques: the MPC-source code for an if-statement is produced by appending the straight-line (MPC) code for the else-arm onto the straight-line (MPC) code for the then-arm, then adding the conditional, and the multiplexer to select the correct values. Due to single assignment, variables used at the if-statement test are unmodified, and are referenced in the comparison expression (CMP) that precedes MUX, where the ϕ nodes capture exactly the arguments of MUX. [6] For example, consider the if-statement in lines 9-28 in Fig. 1(b). The ϕ nodes capture the values of x and y; if control took the then-arm, then x and y would be x2 and y2 respectively, otherwise x and y would be x1 and y1.

In our example, the resulting MPC-source program is shown in Fig. 1(c). We point out that MPC-source can be mapped one-to-one to standard straight-line MPC; the only difference is that when a block is repeated multiple times in straight-line MPC, it is replaced by a for-loop in MPC-source. Following standard MPC compilers methodology, e.g., [BNP08; Fra+14], the actual MPC program unrolls all loops, and loop induction variables become constants.

To make the above mapping explicit, we use *pseudo ϕ-nodes*. To better understand the use of these nodes, let's focus on lines 5, 6 and 12 in Fig. 1(c) at the beginning of each one of the loops; these lines do not encapsulate an if-then-else construct. Instead, they select variable values—at the first iteration, the value comes from outside the loop, and at every subsequent iteration the value comes from the previous iteration of the loop. When translated into straight-line code, these lines disappear because corresponding values are directly used as inputs to the gates. To highlight that these lines are only here to enable loops, and, that these do not get translated into a MUX, we refer to them as *pseudo ϕ-nodes* in text and denote them with $?:$ instead of ϕ.

Looking ahead (cf. §4) the benefit of doing the analysis over MPC-source rather than straight-line code will be that there are significantly fewer variables in the resulting integer/linear program.

3.3 Control-flow Structure of MPC-source

The main reason why most, if not all, MPC compilers use straight-line code as their (intermediate) source representation is that it exhibits a very simple control flow structure. Despite having loops for more compact

[6] MUX is the multiplexer gate that is common in MPC compilers: on input of values (v_0, v_1) and a selection bit $b \in \{0,1\}$, it returns v_b. In our case b is result of the CMP and (v_0, v_1) are arguments of ϕ node.

representation, MPC-source also exhibits simple control-flow structure, which, as we show, facilitates program analysis. Specifically, the program consists of straight-line *blocks* nested within each other. Fig. 3(a) illustrates the block structure of MPC-source.

Each block B, except for the outermost one, is a for-loop block:

$$n_0 \rightarrow n_2 \rightarrow \ldots n_k \longrightarrow n_0$$

Here $n_0, n_1, \ldots$ denote statements in B, short arrows (i.e., $\rightarrow$) denote *forward* control-flow edges in B, and long arrows (i.e., $\longrightarrow$) denote the *back* edge from the last node $n_k \in B$ to the entry node $n_0 \in B$. The node n_0 is special in MPC-source, because it is a *control merge* node. There are two incoming edges into n_0: a forward edge $n' \rightarrow n_0$ where n' is the node in B's enclosing block B' that immediately precedes B, and the back edge $n_k \longrightarrow n_0$.

For example, consider the statement "rem1 = (j == LEN-1) ? rem0 : rem5;" in Fig. 1(c). In the first iteration of the loop, it chooses the value of rem1—this is the value of rem0 in our case, and at every subsequent iteration it chooses the values resulting from the previous iteration—which is the value of rem5 in our case. Node n_k is special as well because it is a *control split* node —there are two outgoing control-flow edges from n_k, a forward edge $n_k \rightarrow n''$, where n'' is the node in B' that immediately succeeds B, and the back edge $n_k \longrightarrow n_0$. The graph below shows the nested structure (it omits the back edge for clarity):

$$\underbrace{\ldots n' \rightarrow \overbrace{n_0 \rightarrow n_2 \rightarrow \ldots n_k}^{B} \rightarrow n'' \ldots}_{B'}$$

3.4 Reaching Definitions over MPC-source

We are interested in Reaching Definitions over MPC-source, because the simple control-flow structure of MPC-source discussed above, as opposed to general IMP-style code, makes Reaching Definitions a very powerful tool. In particular, unlike general IMP programs, in MPC-source programs a def-use chain (d, u) entails that d *always reaches* u due to the simpler control-flow structure of MPC-source programs. Examples of def-use chains in the MPC-source program in Fig. 1(c) are (5,14) (the definition of x1 at line 5 reaches the use at line 14), and similarly (13,14). As another example, the MUX statement at line 25 is a definition of x3 and the statement at line 5 is a use of x3. We will be using def-use chains to calculate the total cost of running an MPC-source program and reason about conversions (see also discussion about optimal conversion placement below).

3.5 Statement Weights

Since MPC-source has loops, in order to accurately capture execution cost, we must assign weights to statement in the MPC-source control-flow graph. (As discussed in the following section, certain edges that are necessary for the definition of our IP are also assigned weights.) The weights correspond to the number of times a statement/edge executes. Once again, the simple structure of our MPC-source representation gives the solution: unlike general IMP-style source-code, in MPC-source it is straight-forward to assign those weights because there are no if-then-else statements, and therefore no need to estimate the number of times control may go through one branch relative to the other (the standard approach is to assume equal probability of execution of each branch). The weight w_n of statement n is the product of the bounds of all loops "around" n: $b_1 \cdot b_2 \cdot \ldots \cdot b_k$ where b_1 stands for the bound of the outermost loop, and b_k for the bound of the innermost loop enclosing n. For example, w_{13} in Fig. 1(c) is $2\mathsf{LEN} \cdot \mathsf{LEN} = 2\mathsf{LEN}^2$.

3.6 Optimal Conversion Placement

Different protocols use different sharings. To stitch such protocols together, we need share conversion. In linearized MPC (where all loops are unrolled) placing such conversions is straight-forward: always convert to what the next protocol needs (if the protocol is the same do not convert). However a challenge in using MPC-source, where loops are present, is when a node is part of a loop whose output needs to be converted. For example, consider a definition that is computed before a loop and is used inside the loop. It is most beneficial to place the conversion before the loop. In this section we describe how to identify the optimal such conversion point to minimize the total cost. This allows us to use the benefits of working with the condensed MPC-source without sacrificing cost efficiency due to suboptimal conversion placement.

Consider a def-use chain (d, u). If d computes a value in one share (e.g., Arithmetic) but u uses a different share (e.g., π^{Y}), then the value computed at d must be converted to the share required at u. We must place conversions in such a way that: (1) each execution path from d to u executes the required conversion, and (2) the total cost of executing the required conversion(s) is minimal; we note that the cost of a single conversion operation is fixed, however, the total cost depends on where, i.e., on what CFG edge, we place the conversion operation. We define $min\_cut(d, u)$[7] where it is least costly to place a conversion of the value computed at d on the way to u. Next, we describe how to find $min\_cut(d, u)$.

We begin with the definition of necessary terms. Let the *closest enclosing block of* n_i *and* n_j be the innermost block B such that $n_i \in B_i$ and $n_j \in B_j$ and both B_i and B_j are nested, immediately or transitively, in B. Trivially, a block is nested in itself. An edge $e = n_1 \to n_2$ is said to be in block B, denoted as $e \in B$, if: either 1) $n_1 \in B$, or 2) $n_2 \in B$, or $n_1 \in B_1$, $n_2 \in B_2$ and B_1 and B_2 are immediately enclosed in B.

To compute the $min\_cut(d, u)$, there are two cases. Case 1 is when d precedes u, i.e., there is a sequence of forward edges from d to u. We call these *forward def-use chains*. In this case, $min\_cut(d, u)$ is the *first edge* e in the sequence of forward edges from d to u such that e is in the closest enclosing block B of d and u. Clearly, the cost of such edge e, w_e, is the number of times B executes. For example, consider def-use chain (14,17) in the MPC-source program in Fig. 1(c). The closest enclosing block of lines 14 and 17 is the inner `for`-loop; the min-cut edge is edge $14 \to 15$, the first in the forward sequence from 14 to 15. As another example, consider def-use chain (17,21). The closest enclosing block of both lines 17 and 21 is the outer `for`-loop. The min-cut edge is the edge from 17 to 21, which executes LEN number of times, and as we mentioned earlier, this entails that it is least costly to place a conversion at 21 rather than at 17.

Case 2 arises when u precedes d, i.e., there is a sequence of forward edges from u to d and the path from d to u goes through a back edge (see also Remark 1.) We call these chains *backward def-use chains*. In this case, it follows directly from the Reaching Definitions analysis and the structure of MPC-source that $min\_cut(d, u)$ is precisely the back edge of the closest enclosing block B of d and u. The cost of such edge e, w_e is $N - 1$ where N is the number of times B executes. In our running example, $min\_cut(25, 5)$ is precisely the back edge of the outer `for`-loop. This edge executes $2\mathsf{LEN} - 1$ times, which is exactly the minimal number of conversions one would need if the MUX at Line 25 of Figure 1.(c) computed x using π^{Y} but it used π^{A} for processing Line 5.

One intuition to the $min\_cut(d, u)$ is as follows: its weight captures the number of distinct statements `st` in the linearized MPC that map to d, such that `st` is used by a use that maps to u.

Remark 1. Note that in any execution *use*s always succeed *def* (It doesn't make sense to use something that isn't defined yet). Our notion of u preceding d and *backward def-use chains* to refer to backward edge in MPC-source CFG is a feature of the MPC-source representation. This backward edge always occurs because of a *pseudo*-ϕ node and disappears in translation to linearized code.

Remark 2. We conclude this section with an observation on backward chains, which will play a role in defining and solving the optimal protocol assignment problem. Backward chains exhibit the following property: each (d, u) is such that d's block is nested in u's block, and u is precisely the pseudo ϕ-node at the beginning of the block. (Let x be the variable defined at d. Suppose u was a use of x other than the pseudo ϕ-node. Since the use of x at u precedes the definition at d, at the first iteration of u's loop, x would come from outer scope. Therefore, SSA would have to merge the two definitions of x into the pseudo ϕ-node, thus creating an earlier definition of x. A subsequent use would refer to the definition at the ϕ-node.)

[7] Note that here *min cut* is slightly different from classical max-flow/min-cut. We want to find min cut on the graph of a single def-use chain.

4 The Optimal Protocol Assignment Problem

In this section we provide formal definitions of the optimal protocol assignment problem (OPA) and in §5 we present our efficient solver. Before defining the problem, we first establish some useful notation and terminology that we will use throughout the section.

Notation and terminology:

(IMP-)source code: This is the starting point of our compiler. It is standard programming language code for an imperative language such as IMP. We denote it by S. All loops have a known upper bound on their iterations.

MPC-source code: The output of our compiler on some source code S. We denote the compiler by $C_{\mathrm{MPC}}(\cdot)$. The compiler removes if-statements and ϕ-nodes, and adds MUX-statements in their place. MPC-source contains for-loops with known bounds.

Block B of MPC-source: Sequence of assignment statements or blocks (in case of for-loop nesting) enclosed in a for-loop.

(IMP-)SSA-code: this is the output of SSA on some source-code S. We will denote it as $C_{\mathrm{SSA}}(S)$. This is an intermediate representation between (IMP-)source and MPC-source.

Linearized-code : $\mathsf{Linear}(S)$: This is the linearization of some MPC-source $C_{\mathrm{MPC}}(\cdot)$. It contains no loops, only straight-line code of assignment statments. The corresponding CFG of this would be simply a straight line. We refer to statement in $\mathsf{Linear}(S)$ as *simple statements* and denote them as $\mathtt{st}$. Since the corresponding CFG is a line we often refer to simple statements as *nodes in (the CFG of)* $\mathsf{Linear}(S)$.

Informally, OPA seeks, given source code for the task the parties wish to securely perform, the best possible combination of MPC modules, i.e., the combination that minimizes a well defined cost function. We stress that existing works attack OPA in a heuristic fashion; to our knowledge, ours is the first work that devises a systematic model and uses it to provide provably optimal solutions—under mild and natural assumptions—to OPA via an automated efficient solver.

There are several parameters that affect the quality of a protocol assignment, and therefore the performance of the resulting hybrid MPC protocol. One of the most important is the cost model, which, informally, specifies the cost of each MPC protocol for computing each statement of the IMP-MPC program. A second important parameter is scheduling. In particular, some protocols are more friendly to amortization/parallelization than other protocols which means that even though protocol X might be preferable to protocol Y for a single statement $\mathtt{st}$—e.g., a multiplication gate—when multiple copies of $\mathtt{st}$ are computed in parallel—Y might be overall preferable to X. For example, on a high-bandwidth/low-latency network (e.g., a LAN), Yao's protocol is faster when computing an (individual) equality-check gates, but when multiple equality gates are computed in parallel, the optimized GMW protocol π^{B} overtakes π^{Y} (this was demonstrated in [DSZ15] and is confirmed in our experiments in §7.3.) We defer the treatment of scheduling to §6.

4.1 The cost model

Coming up with a good measure of the cost is an interesting problem in itself. There is no universally applicable optimal metric and such choice is usually influenced by a program's execution environment. For example, in a data center with high speed connectivity between the servers, minimizing run time would take priority and, therefore, run time is a good cost metric. However, in a data constrained setting e.g. mobile phones, minimizing the size of network traffic may be more desirable. In this case, communication size would be a good cost metric.

In this section we devise a generic user-parameterizable cost model for programs that will be used in the definition of OPA. Informally, the cost model consists of assigning weights, i.e., costs, to different protocols and to conversions of sharings. This is similar to the cost model devised in [KSS14; Cha+17; Büs+18]; however, as we discuss in Section 5, our utilization and application of the cost model is qualitatively different than that of [KSS14] and this will allow us to compute optimal assignments in polynomial time

Let $St = \{\mathtt{st}_1, \ldots, \mathtt{st}_\ell\}$ be the ordered sequence of statements in $\mathsf{Linear}(S)$, and let $\Pi = \{\pi_1, \ldots, \pi_m\}$ be (a set of) multi-party protocols and let $\Sigma = \{\sigma_1, \ldots, \sigma_q\}$ be (a set) of secret sharing schemes (in typical scenarios such as [DSZ15; MR18; Cha+17; Büs+18] $q = m$.) Note that sharings and protocols are very different objects: A protocol is a collection of interactive algorithms to be executed among multiple parties, whereas a sharing scheme is a way to encode/distribute messages (typically protocol inputs and outputs) among those parties. Additionally, although in the literature, protocols are assigned a unique sharing scheme, this does not need to be the case. Therefore, for most generality, in the following we give the definition of the cost model for arbitrary sets of protocols and sharings.

The cost model $\mathcal{C}$ takes into account running each node/simple-statement, plus the cost of conversions between sharings. Formally a cost model $\mathcal{C}$ for a given (St, Π, Σ) is a set containing the following $\ell \cdot m + q^2$ elements:

- For each $(i, j) \in [\ell] \times [m]$: the triple $(\mathtt{st}_i, \pi_j, c^{\pi_j}_{\mathtt{st}_i}) \in B \times \Pi \times \mathbb{Z}_{\geq 0}$, where intuitively, $c^{\pi_j}_{\mathtt{st}_i}$ corresponds to the cost of emulating in a flow statement $\mathtt{st}_i$ with protocol π_j.
- For each $(i, j) \in \Sigma^2$: the triple $(\sigma_i, \sigma_j, c^{\sigma_i 2 \sigma_j}) \in \Sigma \times \Sigma \times \mathbb{Z}_{\geq 0}$, where intuitively, $c^{\sigma_i 2 \sigma_j}$ is the cost of securely converting a sharing according to scheme σ_i into a sharing according to σ_j.

For brevity, and without loss of generality, whenever the sequence St, and set Π are clear from the context we might use $c^{\pi_j}_{\mathtt{st}_i}$ and $c^{\sigma_i 2 \sigma_j}$ instead of the setup of triples. Note that those costs are generic, in the sense that they may be instantiated towards minimization of run time, or towards minimization of data transfer. Furthermore, in all existing works on protocol mixing—including ours—each protocol π_i is associated with a single sharing scheme σ_i; in such cases, in slight abuse of notation, we will denote the conversion cost from σ_i to σ_j as $c^{\pi_i 2 \pi_j}$ (instead of $c^{\sigma_i 2 \sigma_j}$). In fact, to further simplify our notation and consistently with the ABY notation, for the three ABY protocol $\pi^{\mathtt{A}}, \pi^{\mathtt{B}}$, and $\pi^{\mathtt{Y}}$, and for $X, Z \in \{\mathtt{A}, \mathtt{B}, \mathtt{Y}\}$ we will use $c^{X 2 Z}$ to denote the conversion cost $c^{\pi^X 2 \pi^Z}$ from the sharing corresponding to π^X (which we will refer to as Sharing X) to the sharing corresponding π^Z (which we will refer to as Sharing Z).

Generalized Cost Model: Amortization and Parallelization The above cost model does not account for the benefits of amortization and parallelization, and it therefore applies only to linearized code. Therefore, in the following we refer to as the *simple (or linearized) cost model*. The OPA definition and solver from Sections 4.2 and 5, respectively, are actually for linearized MPC. However, in Section 6 we extend our treatment to natural schedulers and show how to (provably) optimally take advantage of amortization for such schedulers. In fact, our implementation and benchmarks do use this scheduler. For completeness, we discuss below how to generalize the cost model to account also for amortization.

To derive a *generalized cost model* we modify the simple cost model as follows: every triple of the type $(\mathtt{st}_i, \pi_j, c^{\pi_j}_{\mathtt{st}_i})$ is generalized to a triple $(\mathtt{st}_i, \pi_j, f_{c^{\pi_j}_{\mathtt{st}_i}}(\cdot))$, where $f_{c^{\pi_j}_{\mathtt{st}_i}} : \mathbb{N} \to \mathbb{Z}_{\geq 0}$ is the amortized execution cost function, which on input $\ell \in \mathbb{N}$ outputs the amortized cost $f_{c^{\pi_j}_{\mathtt{st}_i}}(\ell)$ of computing ℓ parallel copies of $\mathtt{st}_i$ with protocol π_j. Similarly, every triple of the type $(\sigma_i, \sigma_j, c^{\sigma_i 2 \sigma_j})$ is replaced by a triple of the type $(\sigma_i, \sigma_j, f_{c^{\sigma_i 2 \sigma_j}}(\cdot))$, where $f_{c^{\sigma_i 2 \sigma_j}} : \mathbb{N} \to \mathbb{Z}_{\geq 0}$ is the amortized conversion cost function, which on input $\ell \in \mathbb{N}$ outputs the amortized cost $f_{c^{\sigma_i 2 \sigma_j}}(\ell)$ of converting ℓ sharings according to σ_i into sharings according to σ_j. Using the same simplified notation as above, for $X, Z \in \{\mathtt{A}, \mathtt{B}, \mathtt{Y}\}$ we will use $f_{c^{X 2 Z}}$ to denote the function $f_{c^{\pi^X 2 \pi^Z}}$ from the sharing corresponding to π^X to the sharing corresponding π^Z. Naturally the costs of the simple model corresponds to the output of the above functions on input $\ell = 1$.

4.2 OPA for Linearized MPC

Having specified the (simple) cost model $\mathcal{C}$ we can now give a formal definition of the OPA problem. Here we discuss the OPA problem for linearized MPC, which we term *linearized OPA*[8] for which we give an efficient solver in the following section. The more general (non-linearized) case is then treated in §6.

To define linearized OPA we first need to introduce the notion of a *protocol assignment*. Informally, a protocol assignment is defined on the sequence $St = \{\mathtt{st}_1, \ldots, \mathtt{st}_\ell\}$ which is the CFG of $\mathsf{Linear}(S)$; it specifies what protocol should be assigned to each statement (node) $\mathtt{st}_i$. More concretely, a protocol assignment PA

[8] Wherever clear from the context we might drop the adjective linearized and refer to the problem as OPA.

is a sequence of pairs of the type $(\mathtt{st}_1, \pi_1), \ldots, (\mathtt{st}_{|St|}, \pi_{|St|})$, where $(\mathtt{st}_i, \pi_j) \in \mathsf{PA}$ means that statement $\mathtt{st}_i$ is assigned protocol π_j.

Clearly, the execution cost includes the sum of the costs of individual statements $\mathtt{st}_i \in \mathsf{Linear}(S)$. However, we must take into account conversion cost—if PA assigns protocol π^X to $\mathtt{st}_i$, which defines variable x, and it assigns protocol π^Z to $\mathtt{st}_j$ which uses x, then PA entails conversion of x from Sharing X to Sharing Z. Formalizing the above is somewhat tricky as we need to know usage dependencies between the statements to place conversion points. Recall that def-use chains are pairs of the form (d, u) where d and u are nodes in the control-flow graph and u uses d. We need to place share conversion of definition d if there is at least one use u that requires it. Importantly, since we consider $\mathsf{Linear}(S)$, each d executes *exactly once* and therefore, a conversion can be placed immediately after d is executed. Informally, the execution cost is

$$\textstyle\sum_{\mathtt{st}} c^{\pi}_{\mathtt{st}} + \sum_d c^{\pi_i 2 \pi_j}$$

where the first summation term accounts for the execution cost of all program statements, per the protocol π assigned by PA to $\mathtt{st}$, and the second term accounts for necessary conversions: as stated earlier, a conversion at d is necessary if at least one use of d is assigned a different protocol. Below, we formally define the cost function that captures execution and conversion costs.

- Let integer variables $a^{(\mathtt{st}_i, \pi_j)} \in \{0, 1\}$ denote whether $\mathtt{st}_i \in \mathsf{Linear}(S)$ is assigned protocol π_j: $a^{(\mathtt{st}_i, \pi_j)} = 1$ if $(\mathtt{st}_i, \pi_j) \in \mathsf{PA}$; $a^{(\mathtt{st}_i, \pi_j)} = 0$ otherwise.
- Let integer variables $x^{(\mathtt{st}_i, \pi_j, \pi_k)} \in \{0, 1\}$ denote whether protocol assignment PA entails conversion of the definition at node $\mathtt{st}_i$ from (the sharing associated with) protocol π_j into protocol π_k. $x^{(\mathtt{st}_i, \pi_j, \pi_k)} = 1$ if it entails conversion, that is, there is *at least one use* of the variable defined at $\mathtt{st}_i$ that requires π_k. $x^{(\mathtt{st}_i, \pi_j, \pi_k)} = 0$ otherwise.

  More precisely, let statement $\mathtt{st}_i$ define variable x. Protocol assignment PA entails conversion of the definition at node $\mathtt{st}_i$ from π_j into π_k if and only if there exist node $\mathtt{st}_l$ that uses x and

  $$(a^{(\mathtt{st}_l, \pi_k)} - a^{(\mathtt{st}_i, \pi_k)}) \cdot a^{(\mathtt{st}_i, \pi_j)} = 1$$

  The above equation (which is linear if and only if $m = 2$) states that the use statement $\mathtt{st}_l$ is assigned π_k by PA (we have $a^{(\mathtt{st}_l, \pi_k)} = 1$), while the definition at statement $\mathtt{st}_i$ is assigned π_j (we have $a^{(\mathtt{st}_i, \pi_k)} = 0$ and $a^{(\mathtt{st}_i, \pi_j)} = 1$) .

Therefore, the OPA problem becomes: find protocol assignment PA and values of variables $a^{(\mathtt{st}_i, \pi_j)} \in \{0, 1\}$ and $x^{(\mathtt{st}_i, \pi_j, \pi_k)} \in \{0, 1\}$ that minimize the objective function:

$$\textstyle\sum_{\mathtt{st}_i, \pi_j} a^{(\mathtt{st}_i, \pi_j)} \cdot c^{\pi_j}_{\mathtt{st}_i} + \sum_{\mathtt{st}_i, \pi_j, \pi_k} x^{(\mathtt{st}_i, \pi_j, \pi_k)} \cdot c^{\pi_j 2 \pi_k} \quad (1)$$

subject to constraints

$$\sum_{\pi_j \in \Pi} a^{(\mathtt{st}_i, \pi_j)} = 1 \text{ for each node } i \quad (2)$$

and

$$x^{(\mathtt{st}_i, \pi_j, \pi_k)} \geq (a^{(\mathtt{st}_l, \pi_k)} - a^{(\mathtt{st}_i, \pi_k)}) \cdot a^{(\mathtt{st}_i, \pi_j)} \quad (3)$$

for each def-use chain $(\mathtt{st}_i, \mathtt{st}_l)$.

The first term in the summation captures statement execution cost, and the second term captures conversion cost. Note also, that we simplify the problem by assuming that each statement is assigned exactly one protocol. [9] The assumption renders the problem cleaner. Specifically, $a^{(\mathtt{st}_l, \pi_k)} - a^{(\mathtt{st}_i, \pi_k)} = 1$ implies conversion from π_i at the definition to a π_k at the use. If we allowed that a statement is assigned more than one protocols, i.e., $\sum_{\pi_j \in \Pi} a^{(\mathtt{st}_i, \pi_j)} \geq 1$, then it would not be straightforward to capture conversion at the definition: as more than one protocol at the definition can be used to convert to the protocol required at the

[9] In some cases, it may be beneficial to assign more than one protocol, e.g., $\pi^{\mathtt{Y}}$ and $\pi^{\mathtt{A}}$ to the same statement, and perform the computation with each protocol.

use, we would need to take the convert from the available protocol with minimal conversion cost to π_k. In the case of 2 protocols, which is our goal in this paper, we can relax this assumption.

We say that protocol assignment PA *induces variable assignments a and x* when those assignments satisfy constraints (2) and (3).

The above integer program is non-linear if we allow for arbitrary protocols, but becomes linear if we restrict it to two protocols, i.e., $m = 2$. For notational simplicity, we give the definition of the problem for $\pi_i = \pi^{\mathtt{Y}}$ and $\pi_i = \pi^{\mathtt{A}}$, i.e., the (optimized) Yao and Arithmetic protocol from the ABY framework. This is without loss of generality, and our treatment can be trivially applied to any combination of two protocols. We further simplify notation by using $a^{\mathtt{st}_i}$ to denote (the indicator variable) that PA assigns $\pi^{\mathtt{A}}$ to $\mathtt{st}_i$, and $y^{\mathtt{st}_i}$ to denote that it assigns $\pi^{\mathtt{Y}}$ to $\mathtt{st}_i$. We use $x^{\mathtt{st}_i}$ to denote that the definition at $\mathtt{st}_i$ requires Y2A conversion, and $z^{\mathtt{st}_i}$ to denote that $\mathtt{st}_i$ requires A2Y conversion.

The (2-protocol, linearized) OPA problem becomes: find a protocol assignment PA that minimizes

$$\begin{gathered}\textstyle\sum_{\mathtt{st}_i \in St} (a^{\mathtt{st}_i} \cdot c^{A}_{\mathtt{st}_i} + y^{\mathtt{st}_i} \cdot c^{Y}_{\mathtt{st}_i}) \\ + \\ \textstyle\sum_{\mathtt{st}_i \in St} (x^{\mathtt{st}_i} \cdot c^{A2Y} + z^{\mathtt{st}_i} \cdot c^{Y2A})\end{gathered}$$

where

$$a^{\mathtt{st}_i} + y^{\mathtt{st}_i} \geq 1 \text{ for each node } \mathtt{st}_i$$

and

$$\begin{aligned} x^{\mathtt{st}_i} &\geq a^{\mathtt{st}_l} - a^{\mathtt{st}_i} \text{ for each def-use } (\mathtt{st}_i, \mathtt{st}_l) \\ z^{\mathtt{st}_i} &\geq y^{\mathtt{st}_l} - y^{\mathtt{st}_i} \text{ for each def-use } (\mathtt{st}_i, \mathtt{st}_l)\end{aligned}$$

From now on, we will denote this problem as $IP_{\mathsf{Linear}(S)}$.

For our purposes constraint $x^{\mathtt{st}_i} \geq a^{\mathtt{st}_l} - a^{\mathtt{st}_i}$ is equivalent to $x^{\mathtt{st}_i} \geq (a^{\mathtt{st}_l} - a^{\mathtt{st}_i}) \cdot y^{\mathtt{st}_i}$.

In §5 we show how to efficiently solve the above linear integer program, as well as a related more efficient one defined directly on MPC-source programs. Then in §6 we extend our treatement to a natural class of non-linearized (i.e., parallelized) MPC-source programs. The extension to $m > 2$ is an interesting direction for future research.

5 Solving the Linearized OPA

We now describe our efficient linearized-OPA solver for two protocols ($m = 2$). Recall a solution to linearized OPA is a solution to $IP_{\mathsf{Linear}(S)}$ defined in the previous section, which in turn describes an optimal protocol (and share conversion) assignment for the linearized (straight-line) code. Formally, in this section we prove the following theorem:

Theorem 1. *Let $IP_{\mathsf{Linear}(S)}$ be the integer program corresponding to the linearized OPA problem defined above, and let $LP_{\mathsf{Linear}(S)}$ be its LP relaxation. The optimal solution to $LP_{\mathsf{Linear}(S)}$ is integral, and therefore also the optimal solution to $IP_{\mathsf{Linear}(S)}$.*

In a nutshell, the above theorem is proved by showing that the constraint matrix of $LP_{\mathsf{Linear}(S)}$ satisfies a property known as total unimodularity (cf. Definition 2); a theorem from combinatorial optimization implies then that its solution is in fact integral [Sch03].

We remark that although theoretically interesting, and against what was previously conjectured, having an efficient (polynomial) solver for $IP_{\mathsf{Linear}(S)}$ does not necessarily yield a practical MPC protocol mixer. Indeed, since in linearized MPC loops are entirely unrolled, the corresponding representation might end up having millions of statements and therefore millions of constraints, hindering scalability of the $LP_{\mathsf{Linear}(S)}$ solver.

Therefore, we devise a solver that solves a smaller integer program over MPC-source, denoted by $IP_{C_{\mathrm{MPC}}(S)}$. We stress that existing frameworks compute protocol assignments, at most as optimal as a solution to $IP_{C_{\mathrm{MPC}}(S)}$; indeed, in ABY, the manual protocol assignment is made on the source code, which is essentially MPC-source. In fact, in Theorem 3 we prove that this is always the case under natural conditions on the optimal assignment computed by $IP_{\mathsf{Linear}(S)}$. Since $\mathtt{st}$ nodes in $\mathsf{Linear}(S)$ that map to the same n in $C_{\text{MPC}}(S)$ appear in *identical* contexts of execution in different iterations of the loop, we conjecture that the above statement holds even unconditionally, i.e., if a protocol assignment is optimal in one context, the same assignment will be optimal in the other. We note in passing that although making the treatment more involved, devising such a scalable solver is essential for deriving a practical solution to the problem. Additionally, following the same structure of the proof of unimodularity of the constraint matrix of $IP_{C_{\text{MPC}}(S)}$, we can directly devise a proof of unimodularity of the constraint matrix of $IP_{\mathsf{Linear}(S)}$, thereby proving the result above.

The remainder of this section is organized as follows: In §5.1 we describe $IP_{C_{\text{MPC}}(S)}$, where §5.1 describes the parameters of the $IP_{C_{\text{MPC}}(S)}$ integer program, and §5.1 and §5.1 describe the constraints and objective function. As in the previous section, to keep notation simple we focus on the two protocols, namely arithmetic (π^{A}) and Yao-based (π^{Y}). In §5.2 we prove our main result that due to the structure of $IP_{C_{\text{MPC}}(S)}$ its LP relaxation yields an integral solution; this means that we can use standard efficient LP solvers to solve $IP_{C_{\text{MPC}}(S)}$; finally, in §5.3 we prove that the solution to $IP_{C_{\text{MPC}}(S)}$, under natural conditions, is also a solution to $IP_{\mathsf{Linear}(S)}$. Due to limited space, the proofs have been moved to Appendix D.

5.1 Defining $IP_{C_{\text{MPC}}(S)}$

$IP_{C_{\text{MPC}}(S)}$ is an integer program over MPC-source. It entails a significantly smaller number of variables and constraints, and therefore accepts a more scalable solver. (There are $O(N)$ nodes in MPC-source compared to $O(b^D N)$ nodes in $\mathsf{Linear}(S)$, where b is the maximum loop bound and D is the loop nesting depth.) When no amortization is considered, the costs of executing and converting all $\mathtt{st} \in \mathsf{Linear}(S)$ that map to the same $n \in C_{\text{MPC}}(S)$ is the same. As we show in §5.3, if we constrain $IP_{\mathsf{Linear}(S)}$ to the same $a^{\mathtt{st}}$ and $y^{\mathtt{st}}$ for all $\mathtt{st}$ that map to the same $n \in C_{\text{MPC}}(S)$, the optimal solution of $IP_{C_{\text{MPC}}(S)}$ is the optimal solution of $IP_{\mathsf{Linear}(S)}$ as well.

The Cost Model for $IP_{C_{\text{MPC}}(S)}$

Since we do not have parallelization/amortization, $IP_{C_{\text{MPC}}(S)}$ has a simple cost model as defined in the previous section. Concretely,

(1) c_n^A denotes the cost to run node $n \in C_{\text{MPC}}(S)$ using π^{A}.

(2) c_n^Y denotes the cost to run node n using π^{Y}.

(3) c^{A2Y} denotes the cost to run A2Y conversion.

(4) c^{Y2A} denotes the cost to run Y2A conversion.

Variables and Constraints We follow [Cho+07] to define variables and constraints. Let variables a^n and y^n be integers in the interval $\{0, 1\}$, as in $IP_{\mathsf{Linear}(S)}$ we defined in §4.2. They denote whether node n executes with π^{A} (using Arithmetic sharing) or with π^{Y} (using Yao sharing). $a^n = 1$ if n runs using Arithmetic sharing, and $a^n = 0$ if n runs using π^{Y} sharing. To enforce that each node must execute at least once, we introduce constraint

$$a^n + y^n \geq 1 \qquad (1)$$

Let integer program variable $x_{(d,u)} \in \{0, 1\}$ denote whether (d, u) requires Y2A conversion of x, that is, d computes x using π^{Y} sharing only, but u, which uses x, computes using Arithmetic sharing, and thus requires conversion of x to Arithmetic. Analogously, let $z_{(d,u)} \in \{0, 1\}$ denote whether (d, u) requires A2Y conversion. $z_{(d,u)} = 1$ if it does, and $z_{(d,u)} = 0$ if it does not. Intuitively, the following constraints would account for this:

$$x_{(d,u)} \geq a^u - a^d \qquad z_{(d,u)} \geq y^u - y^d$$

That is, if a^u is 1 but a^d is 0, or in other words d computes using π^{Y}, variable $x_{(d,u)}$ is forced to 1. Later, when we minimize the total cost, we multiply $x_{(d,u)}$ by the weight of (d, u), which is the number of times the min-cut edge of (d,u) executes. Note that if $a^u - a^d$ (or $y^u - y^d$) is -1, then $x_{(d,u)}$ (or $z_{(d,u)}$) would be 0 because of the interval restriction: $x_{(d,u)}, z_{(d,u)} \in \{0,1\}$.

However, a wrinkle arises here. Since there are multiple def-use chains that start at d, the min-cut edge of (d,u) may already cover a different def-use (d,u') yielding constraints

$$x_{(d,u)} \geq a^u - a^d \qquad x_{(d,u')} \geq a^{u'} - a^d$$

too strong: since $x_{(d,u)}$ already covers (d,u'), if both (d,u) and (d,u') require conversion, it is sufficient to perform conversion along the min-cut edge of (d,u); conversion along the min-cut edge of (d,u') would be redundant. (Clearly, there may be more than one uses for each def, but there is only a single def per use, due to the SSA property.) We therefore introduce the notion of *subsumption.*

Definition 1. *Def-use chain (d,u) subsumes def-use chain (d,u'), denoted $(d,u) \supseteq (d,u')$, if and only if min_cut(d,u) dominates u', or in other words, all paths from d to u' go through min_cut(d,u).*

Intuitively, subsumption means that conversion of d at the min-cut edge of (d,u) covers (d,u') as well, and there is no need to introduce conversion at the min-cut edge of (d,u'). There is no natural case for subsumption in our running example. For the sake of argument, assume there is a use of `rem3` defined at line 14, in the outer loop at line 20. Then there are def-use chains (14,15) and (14,20). *min_cut*$(14,15)$ is edge $14 \to 15$, and *min_cut*$(14,20)$ is edge $17 \to 20$. However, (14,15) subsumes (14,20). Assuming that both uses, 15 and 20, require conversion, then placing a conversion at $14 \to 15$ covers (14,15) and (14,20). If 15 does not require conversion but 20 does, then placing a conversion at the less costly edge $17 \to 20$ suffices.

The above definition gives rise to a directed graph with nodes for all def-use chains (d,u) for d, and edges due to subsumption: there is an edge from (d,u) to (d,u') if and only if $(d,u) \supseteq (d,u')$. Strongly connected components (SCCs) in this graph imply several (d,u)'s with the same min-cut edge. We therefore collapse SCCs into equivalence classes with a *representative* e—each equivalence class is covered by a min-cut edge e—and extend the ordering to the representative edges e. For example, suppose we have a chain $d \to u_1 \to u_2 \to n$ in one block, where u_1 and u_2 are uses of d. Suppose we have $n \to u_3$ where u_3 is a use in the immediately enclosing block. (d,u_1) and (d,u_2) are in the same equivalence class with representative edge $d \to u_1$, and (d,u_3) is in another class, with representative edge $n \to u_3$. We have $d \to u_1 \supseteq n \to u_3$.

We now introduce a new set of constraint variables, x_e^d and z_e^d, similar to variables $x_{(d,u)}$ and $z_{(d,u)}$ we introduced earlier. In the integer program we use only variables x_e^d and z_e^d. $x_e^d \in \{0,1\}$ denotes whether there is an Y2A conversion of the variable defined at d on edge e.

Therefore, our constraints become:

$$x_{e_1}^d + \cdots + x_{e_k}^d \geq a^u - a^d$$

where e_k is the representative of (d,u)'s equivalence class, and $e_i \supseteq e_{i+1}$ for $1 \leq i \leq k-1$. These constraints state that if (d,u) requires conversion from Arithmetic to Yao's protocol, it is sufficient to execute that conversion along a min-cut edge for some (d,u') that subsumes (d,u), even when that edge is not the min-cut edge for (d,u) itself.

In the above constraint, edge e_k is the representative edge for the equivalence class of (d,u). If (d,u) is a backward chain, then e_k is the back edge in u's block, and $e_1, \ldots e_{k-1}$ are forward edges totally ordered by subsumption. If (d,u) is a forward chain, then all edges are forward edges and totally ordered by subsumption: $e_1 \supseteq e_2 \cdots \supseteq e_{k-1}$. This structure of constraints that account for conversion helps establish total unimodularity of the constraint matrix, as we detail in the following section.

To summarize, we have constraints that account for conversion from $\pi^{\mathtt{Y}}$ to $\pi^{\mathtt{A}}$:

$$x_{e_1}^d + \cdots + x_{e_k}^d \geq a^u - a^d \qquad (2)$$

and parallel constraints that account for conversion A2Y:

$$z_{e_1}^d + \cdots + z_{e_k}^d \geq y^u - y^d \qquad (3)$$

Objective Function The integer programming problem must find an assignment for variables a^n, y^n, x_e^d and z_e^d that satisfies the above constraints, and *minimizes* the cost of running the program. The total cost is the sum of execution cost and conversion cost:

$$\sum_n (a^n \cdot c_n^A \cdot w_n + y^n \cdot c_n^Y \cdot w_n) \\ + \\ \sum_{d,e} (x_e^d \cdot c^{Y2A} \cdot w_e + z_e^d \cdot c^{A2Y} \cdot w_e)$$

The first summation term models the cost of execution of program statements and is straight-forward. E.g., if n runs using π^{A} then its cost would be c_n^A. The cost of a single run of n is multiplied by w_n, the number of times n executes. In MPC-source w_n is always statically known. The second term models conversion cost and is less straight-forward. It iterates over all d, e pairs where d is a definition and e is a min-cut edge representing some (d, u) (more precisely, an equivalence class of (d, u)'s). w_e is the number of times the min-cut edge e executes. Again, in MPC-source w_e is always statically known. To see the intuition behind the second term, suppose we have two forward def-use chains (d, u) and (d, u') where (d, u) subsumes (d, u') but not the other way around. (d, u)'s representative is min-cut edge e and (d, u')'s representative is e'. The term that accounts for conversions of d (just Y2A), is $x_e^d \cdot c^{Y2A} \cdot w_e + x_{e'}^d \cdot c^{Y2A} \cdot w_{e'}$. If the assignments to a^d and a^u entail conversion, then x^e is 1, and therefore, $x^{e'}$ is 0, thus nullifying term $x_{e'}^d \cdot c^{Y2A} \cdot w_{e'}$, just as expected, since (d, u) subsumes (d, u'). Conversely, if a^d and a^u do not entail conversion, then x^e is 0. If (d, u') does require conversion, we will have $x^{e'} = 1$, thus converting definition d $w_{e'}$ times only, where $w_{e'} < w_e$ since e' lies in an outer loop, and e lies in an inner loop.

Therefore, $IP_{C_{\text{MPC}}(S)}$ is as follows:

Minimize

$$\sum_n (a^n \cdot c_n^A \cdot w_n + y^n \cdot c_n^Y \cdot w_n) \\ + \\ \sum_{d,e} (x_e^d \cdot c^{Y2A} \cdot w_e + z_e^d \cdot c^{A2Y} \cdot w_e)$$

subject to

$$Ax \geq b$$

where vector $x = a^{n_1}, y^{n_1}, a^{n_2}, y^{n_2}, \ldots, x_e^d, z_e^d, \ldots,$ and constraint matrix A consists of rows corresponding to constraints (1), (2) and (3). All entries of A are 0 or ± 1.

5.2 Solving $IP_{C_{\text{MPC}}(S)}$ (and $IP_{\mathsf{Linear}(S)}$) via LP

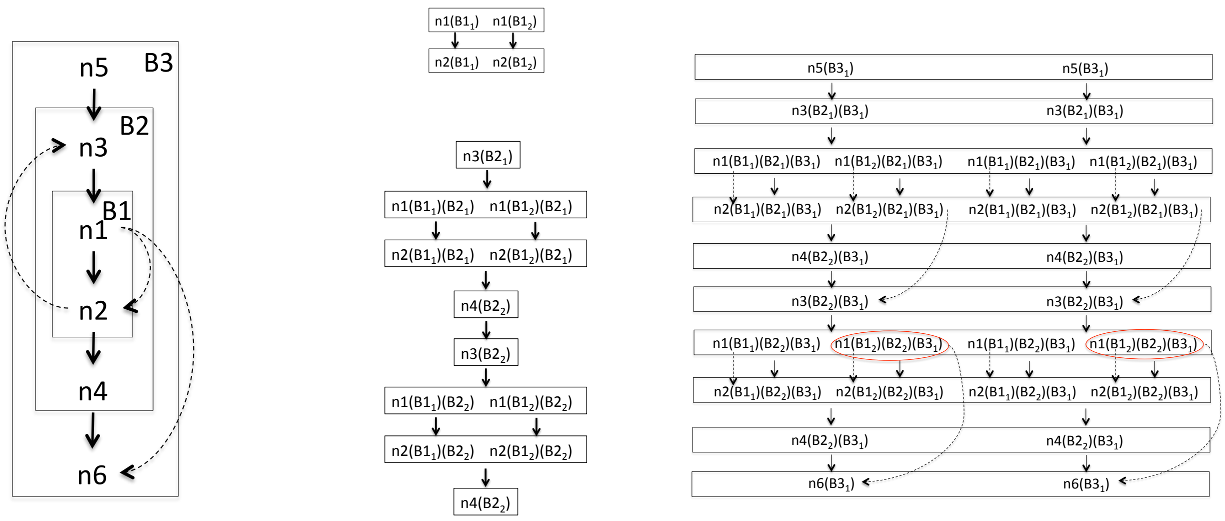
We next prove that the LP relaxation $LP_{C_{\text{MPC}}(S)}$ of $IP_{C_{\text{MPC}}(S)}$ has a totally unimodular constraint matrix and therefore an integral solution (as classical combinatorial optimization results imply, cf. [Sch03]). First, let us recall the definition of total unimodularity.

Definition 2. *A matrix M is totally unimodular if every square submatrix of M has determinant 0, $+1$, or -1. This implies that all entries of M are 0, or ± 1 [Sch03].*

Fortunately, the constraint matrix A in the integer program from §5.1 and §5.1 is totally unimodular. We show this by way of a characterization given by Camion [Cam65] (cf. Appendix D.1.)

Theorem 2. *(Total unimodularity of constraint matrix A.) Let A be the constraint matrix of $IP_{C_{\text{MPC}}(S)}$. For every square Eulerian submatrix of A, $A_J^I : \sum_{i \in I, j \in J} A_i^j \equiv 0 \pmod 4$.*

Remark 3. [On applying our method to three protocols simultaneously] Our approach does not generically extend to 3 or more protocols. The reason is that the direct extension of our IP to $m > 2$ protocols changes the structure of the underlying constraint matrix, in a way that total unimodularity no longer holds. A way to see this is the following: we used constraints of the type $a^u - a^d$ to capture conversions to arithmetic from a different protocol. In the binary ($m = 2$) case, $a^d = 0$ implies that node d was computed in π^{Y}, and therefore, $a^u - a^d = 1$ induces a Y2A conversion. When $m = 3$, $a^u - a^d = 1$ would induce a Y2A *or* a B2A conversion. As in general, $\pi_i 2\pi_j$ and $\pi_{i'} 2\pi_j$ conversions have different costs, devising the corresponding constraints to capture conversions becomes non-trivial, and the matrix is no longer totally unimodular.


n5
B3
B2
n3
B1
n1
n2
n4
n6
n1(B1$_1$) n1(B1$_2$)
n2(B1$_1$) n2(B1$_2$)
n3(B2$_1$)
n1(B1$_1$)(B2$_1$) n1(B1$_2$)(B2$_1$)
n2(B1$_1$)(B2$_1$) n2(B1$_2$)(B2$_1$)
n4(B2$_2$)
n3(B2$_2$)
n1(B1$_1$)(B2$_2$) n1(B1$_2$)(B2$_2$)
n2(B1$_1$)(B2$_2$) n2(B1$_2$)(B2$_2$)
n4(B2$_2$)
n5(B3$_1$)
n3(B2$_1$)(B3$_1$)
n1(B1$_1$)(B2$_1$)(B3$_1$) n1(B1$_2$)(B2$_1$)(B3$_1$)
n2(B1$_1$)(B2$_1$)(B3$_1$) n2(B1$_2$)(B2$_1$)(B3$_1$)
n4(B2$_2$)(B3$_1$)
n3(B2$_2$)(B3$_1$)
n1(B1$_1$)(B2$_2$)(B3$_1$) n1(B1$_2$)(B2$_2$)(B3$_1$)
n2(B1$_1$)(B2$_2$)(B3$_1$) n2(B1$_2$)(B2$_2$)(B3$_1$)
n4(B2$_2$)(B3$_1$)
n6(B3$_1$)

(a) MPC-source with def-use chains (b) Linearization of B1 and B2 (c) Linearization of B3: Parallel(S)

Figure 3. Natural Schedule. There are no backward def-use chains in B1, and therefore B1 is parallelized, executing $n1(B1_1)$ and $n1(B1_2)$ in parallel, as shown at the top of Fig. (b). (We assume each loop has bound 2. $n1(\text{B}1_1)$ denotes the execution of $n1$ in the first iteration of B1, and $n1(\text{B}1_2)$ in the second.) There is a backward def-use chain in B2, $(n2, n3)$ and therefore B2 cannot be parallelized. The two iterations of B2 happen sequentially. There is no backward def-use chain in B3, therefore B3 can be parallelized too, resulting in the final schedule shown in (c). Fig. 3(c) shows concrete def-use chains. There are 8 concrete def-use chains, shown with dashed arrows, that correspond to $(n1, n2)$, and there are 2 def-use chains that correspond to $(n1, n6)$. Conversion due to $(n1, n2)$ is amortized over 4 parallel executions, however conversion due to $(n1, n6)$ is amortized over 2.

5.3 From $IP_{C_{\text{MPC}}(S)}$ to $IP_{\mathsf{Linear}(S)}$

In this section we show that under the assumption that all $\mathtt{st} \in \mathsf{Linear}(S)$ that map to the same $n \in C_{\text{MPC}}(S)$ are assigned the same share, the protocol assignment that minimizes the objective function of $IP_{C_{\text{MPC}}(S)}$ minimizes the objective function of $IP_{\mathsf{Linear}(S)}$ as well.

We define "abstraction" function $\alpha : \mathsf{Linear}(S) \to C_{\text{MPC}}(S)$ and "concretization" function $\gamma : (C_{\text{MPC}}(S) \times C_{\text{MPC}}(S)) \to 2^{\mathsf{Linear}(S)}$ that will help us formalize and establish equivalence (cf. Appendix D.2). Function $\alpha(\mathtt{st})$ returns the node n in $C_{\text{MPC}}(S)$ that $\mathtt{st}$ maps to. Function $\gamma((d, u))$ takes a def-use chain in $C_{\text{MPC}}(S)$, and returns the set of definitions $\mathtt{st}_d$ such that $(\mathtt{st}_d, \mathtt{st}_u)$ is a def-use chain in $\mathsf{Linear}(S)$, and $\alpha(\mathtt{st}_d) = d$, and $\alpha(\mathtt{st}_u) = u$. Intuitively, $\gamma((d, u))$ returns all distinct $\mathtt{st}_d$, such that there are distinct constraints in $IP_{\mathsf{Linear}(S)}$

$$x^{\mathtt{st}_d} \geq a^{\mathtt{st}_u} - a^{\mathtt{st}_d} \text{ s.t. } \alpha(\mathtt{st}_d) = d, \alpha(\mathtt{st}_u) = u$$

and thus

$$x^{\mathtt{st}_d} \geq a^{\alpha(\mathtt{st}_u)} - a^{\alpha(\mathtt{st}_d)} \text{ equiv. } x^{\mathtt{st}_d} \geq a^u - a^d$$

We note that we abuse notation slightly, by using a and y interchangeably in $IP_{\mathsf{Linear}(S)}$ and in $IP_{C_{\text{MPC}}(S)}$.

Theorem 3. *Consider a protocol assignment* $\mathsf{PA} = (a^n, y^n)$ *that minimizes* $IP_{C_{MPC}(S)}$. *If for every pair* $st, st' \in \mathit{Linear}(S)$, $\alpha(st) = \alpha(st') \Rightarrow a^{st} = a^{st'} \wedge y^{st} = y^{st'}$, *then* $a^{\alpha(st)} = a^n$, $y^{\alpha(st)} = y^n$ *minimizes* $IP_{\mathit{Linear}(S)}$.

6 Scheduling and Parallelization

Scheduling specifies the order in which different instructions should be executed and, in particular, which instructions should be executed in parallel. Scheduling and parallelization have been extensively studied in the compilers and parallel programming literature. However, the applicability to MPC of known algorithms and results on loop parallelization, is not well-understood. We conjecture (and leave for future work) that MPC-structure can be exploited to build provably optimal schedules. In this section, we describe a *natural schedule* that targets common patterns occurring in MPC applications. We believe that existing work [Büs+18; BK15], uses essentially the same approach to scheduling, however, we are the first to formally and explicitly describe the schedules.

The original ABY framework takes a greedy parallelization approach: whenever something is parallelizable, assign to the parallel operation the protocol which, when amortized is optimal. Clearly this does not always yield the optimal assignment. More recent versions of the framework [Büs+18] employ heuristics from parallel programming to detect parallelization [Wil+94; IJT91]. Although this might, at times yield a faster execution, there are no guarantees, in general, that the heuristically discovered scheduling is better than no parallelization or full parallelization. In fact, one can construct examples in which the cost of conversion after the parallelized node supersedes the benefits of amortization. For example, a single EQ (equality check) is processed faster with $\pi^{\mathtt{Y}}$ but allows for better amortization when processed with $\pi^{\mathtt{B}}$.

To avoid the ambiguity introduced by scheduling and parallelization, OPA can be parameterized by an explicit scheduler. In the following we describe how we define such a scheduler (§6.1). We describe a *natural parallelization schedule* (§6.2), and what restriction we impose on a given schedule (§6.3). The restriction guarantees that the solution of $IP_{C_{\mathrm{MPC}}(S)}$ is a solution to the $IP_{\mathsf{Linear}(S)}$, and natural parallelization schedules meet the restriction.

6.1 Scheduler

We define schedulers over $\mathsf{Linear}(S)$—recall, these are the linear CFGs corresponding to the linearized MPC. $\mathsf{Linear}(S)$ can be extended to capture parallel execution of the program, by grouping multiple statements into one hyper-node (aka parallel node). All $\mathtt{st}$'s grouped into a parallel node can execute in parallel. $\mathsf{Parallel}(S)$ is the sequence of parallel nodes $P_1 \to P_2 \to \cdots \to P_n$, where P_1 executes before P_2, P_2 executes before P_3, etc. We say that $\mathsf{Parallel}(S)$ is a parallelization of $\mathsf{Linear}(S)$ if and only if for every def-use chain $(\mathtt{st}_d, \mathtt{st}_u) \in \mathsf{Linear}(S)$, $\mathtt{st}_d$ is in hyper-node P_i, $\mathtt{st}_u$ is in hyper-node P_j, and P_i executes before P_j. The restriction is necessary to preserve program correctness—a definition must execute before all its uses.

Definition 3 (OPA-scheduler). *An OPA scheduler* $\mathtt{S}$ *for* $\mathsf{Linear}(S)$ *is a mapping from* $\mathsf{Linear}(S)$ *to a parallelization (schedule)* $\mathsf{Parallel}(S)$*.*

6.2 A Natural Schedule

A natural schedule arises as follows. Assume MPC-source, as shown in Fig. 3(a). If a loop B is such that there is no backward def-use chain that ends in B—and thus, there are no data dependencies from iteration k to iteration $(k+1)$ of B—then we schedule B's iterations in parallel by grouping corresponding nodes into a hyper-node; otherwise, we schedule the iterations sequentially, as in $\mathsf{Linear}(S)$. We call the former case a *parallel loop*, and the latter case a *sequential loop*. For example, the innermost loop $B1$ in Fig. 3(a) is a parallel loop. The schedule of $B1$ is shown at the top of Fig. 3(b). $n1(B1_1)$ and $n1(B1_2)$ are scheduled in the same hyper-node, say P_1, and $n2(B1_1)$ and $n2(B1_2)$ are scheduled in P_2, and P_1 executes before P_2. Loop $B2$ is a sequential loop. The schedule of $B2$ is shown in Fig. 3(b) as well. Since there is a backward def-use chain $B2$'s iterations are scheduled sequentially.

We construct a natural schedule inductively, from the innermost (level 0) towards the outermost (level D) loop. Assume a schedule $S_k : P_1 \to P_2 \dots P_l$ at level k, enclosed in a loop block B with bound b at level $(k+1)$. If B is a parallel loop, then the new schedule S_{k+1} is constructed by grouping together all $P_i(B_1), P_i(B_2), \dots, P_i(B_l)$. $S_{(k+1)}$ is

$$P_1(B_1)\dots P_1(B_b) \to P_2(B_1)\dots P_2(B_b) \to \cdots \to P_l(B_1)\dots P_l(B_b)$$

Conversely, if B is a sequential loop, $S_{(k+1)}$ is constructed by sequencing S_k b times:

$$P_1(B_1) \to \cdots \to P_l(B_1) \to P_1(B_2) \to \cdots \to P_l(B_2) \to \cdots \to P_l(B_b)$$

The final schedule constructed from the MPC-source abstraction in Fig. 3(a) is shown in Fig. 3(c).

We stress that the natural schedule, which we construct in the implementation, is only a step towards a solution. We believe that one can exploit the well-behaved MPC-source representation, and data dependences on MPC-source, to construct provably optimal schedules. We will explore this direction in future work.

6.3 Uniformly Parallel Schedule

In this section, we describe a restriction on parallelization—namely, we consider schedules that have a property we call *uniformly parallelization*. This restriction captures natural schedules, and it also enables computing an optimal protocol assignment that takes advantage of amortized costs. Similarly to §5.3, if we constrain $IP_{\mathsf{Parallel}(S)}$ to the same $a^{\mathtt{st}}$ and $y^{\mathtt{st}}$ for all $\mathtt{st}$ that map to the same $n \in C_{\text{MPC}}(S)$, then the optimal solution of $IP_{C_{\text{MPC}}(S)}$ is the optimal solution of $IP_{\mathsf{Parallel}(S)}$. Below we formalize the uniform parallelization restriction. In Appendix E we argue that a natural schedule as described in §6.2, meets the uniform parallelization restriction, and therefore, the protocol assignment that minimizes $IP_{\mathsf{Linear}(S)}$ minimizes $IP_{\mathsf{Parallel}(S)}$ for a natural schedule.

First, we extend the concretization function γ, to work on $n \in C_{\text{MPC}}(S)$: $\gamma(n) = \{ \mathtt{st} \in \mathsf{Linear}(S) \mid \alpha(\mathtt{st}) = n \}$. We note that we abuse notation by allowing an ill-formed domain of γ. The restriction has two components:

1. All $\gamma(n)$ are uniformly allocated across N hyper-nodes (parallel nodes) in $\mathsf{Parallel}(S)$. That is, each one of the N hyper-nodes contains $\frac{|\gamma(n)|}{N}$ $\mathtt{st}$ nodes. As a result, we can amortize execution costs of the $\mathtt{st}$ nodes based on $\frac{|\gamma(n)|}{N}$, and associate the *same* (potentially amortized) costs c_n^A and c_n^Y with each $\mathtt{st} \in \gamma(n)$. These costs can be extracted from a generalized cost model.

2. All $\gamma((d,u))$ are uniformly allocated across M hyper-nodes. Again, each one of the M hyper-nodes contains $\frac{|\gamma((d,u))|}{M}$ $\mathtt{st}_d$ nodes. As a result, we can amortize conversion costs of the $\mathtt{st}_d \in \gamma((d,u))$ nodes based on $\frac{|\gamma((d,u))|}{M}$, and associate the *same* (potentially amortized) conversion costs $c_{d(e)}^{Y2A}$ and $c_{d(e)}^{A2Y}$—also extracted from the generalized cost model—with each $\mathtt{st}_d \in \gamma((d,u))$, where $e = \mathit{min\_cut}(d,u)$. Notably, the cost of converting d depends on what the min-cut edge e happens to be.

   By Lemma 4 (see Appendix D.2), if (d,u) subsumes (d,u'), then $\gamma((d,u')) \subseteq \gamma((d,u))$, i.e., only a subset of the definitions $\mathtt{st}_d$ are part of def-use chains that end at u''s. Thus, $\gamma((d,u'))$ is amortized over a smaller number of parallel executions, and therefore, individual conversion cost $c_{d(e')}^{Y2A}$ may be higher than individual conversion cost $c_{d(e)}^{Y2A}$. If conversions are not required at (d,u), but they are required at (d,u'), those conversions contribute higher cost, namely $c_{d(e')}^{Y2A}$, than (d,u).

$\mathsf{Linear}(S)$ is an extreme case of a uniformly parallel schedule: all parallel nodes are of size 1, and all costs are the sequential costs.

Theorem 4. *The protocol assignment that minimizes*

$$\begin{gathered}\textstyle\sum_n \left(a^n \cdot c_n^A \cdot w_n + y^n \cdot c_n^Y \cdot w_n\right) \\ + \\ \textstyle\sum_{d,e} \left(x_e^d \cdot w_e \cdot c_{d(e)}^{Y2A} + z_e^d \cdot w_e \cdot c_{d(e)}^{A2Y}\right)\end{gathered}$$

also minimizes

$$\textstyle\sum_{st} \left(a^{st} \cdot c_{st}^A + y^{st} \cdot c_{st}^Y\right) + \sum_{st}\left(x^{st} \cdot c_{st}^{Y2A} + z^{st} \cdot c_{st}^{A2Y}\right)$$

This is argued exactly as in §5.3.

7 Implementation and Benchmarks

In this section we discuss our implementation and experimental results. The section is organized as follows: §7.1 presents an overview of our implementation—the analysis and OPA solver, and §7.2 describes our

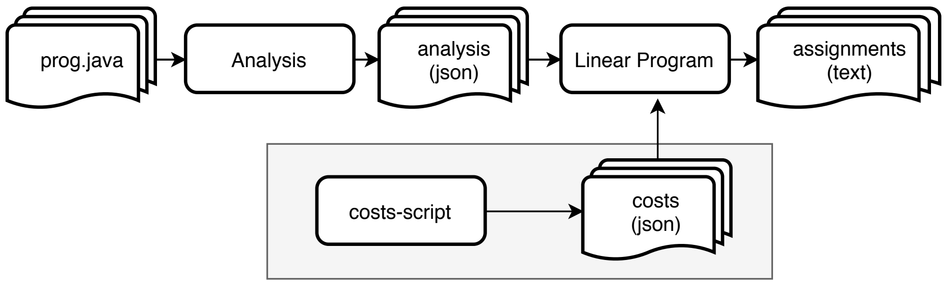


Figure 4. Implementation Overview: The analysis takes a Java program as input and outputs a def-use graph (along with related information). The linear program takes as input analysis information and costs, and outputs the optimal assignment.

Table 1. Running Times of Analysis and Integer Program (rounded to nearest integer, median of 10 executions).

Benchmark	**Lines of Code** Java	MPC-Source	MPC Nodes†	**Time (secs)** Analysis	Integer Program
GCD	36	55	10	18	1
Biometric Matching	55	112	7	19	1
Modular Exponentiation	43	112	19	18	1
Private Set Intersection (PSI)	40	75	2	18	1
Histogram	102	160	24	18	1
MiniONN (MNIST)	196	696	114	23	4
k-means	121	331	36	19	2
DB-Merge (500 + 500)	77	192	26	19	1
DB-Join (50 x 50)	83	189	33	19	1
DB-Join (25 x 200)	103	225	43	19	1
Cryptonets (Square)	103	331	39	19	1

† MPC-Source nodes may translate to several gates, e.g. in the running GCD example, the `MUX` on line 25 is translates to 2 * LEN gates.

experiments. §7.3 details how we calculated costs for the cost model and discusses the implications of our method. §7.4 concludes with a detailed examination of our results, and a comparison with existing works. For brevity, we may refer, collectively, to our implementation of the analysis and OPA solver as the *toolchain* or the *tool*.

7.1 The Toolchain

Our techniques are generically applicable to MPC-source, which can be defined on any high-level language, i.e., any language that can be transformed into IMP-SSA form is a candidate for our analysis. In our experiments we chose Java as the high level language for our system. Following the methodology introduced in the previous sections, we restrict our benchmarks to an IMP-style subset of Java that can be translated to MPC-source. This yields the following restrictions which are standard in MPC compilers [BNP08; SR18; Fra+14]: 1) function calls are statically resolvable, i.e. no polymorphism, 2) there is no recursion, 3) loops have statically known bounds, and 4) arrays have statically known sizes.

Additionally, we restrict data types to unsigned integers (for both scalars and arrays). We note that this restriction does not entail loss of generality. If the underlying compiler supports additional data types, the analysis can easily be extended to handle those data types. The OPA solver itself will remain unchanged. However, costs for operations on the additional data types will have to be collected (the OPA solver needs costs for all operations). In fact, one future research direction is to integrate our toolchain into a feature-rich hybrid protocol compiler such as CGMC-GC [Fra+14].

We used Soot [Val+99] for performing our program analysis. Soot is a popular program analysis framework for Java and Android. It provides an SSA form called *Shimple*. It also provides out-of-the-box support for function inlining, loop detection and basic def-use analysis, which facilitate translation of Java to IMP-SSA, and subsequently to MPC-source.

We used MATLAB's Optimization Toolbox to write a linear program that takes analysis information as input (along with costs) and outputs an optimal mixed protocol assignment for the specified two protocols.

Figure 4 presents an overview of our system. The analysis takes a Java program as input. Using Soot, it transforms the input program into SSA form (Shimple), then inlines the function calls. The program is now transformed into MPC-source. (A mapping from Shimple operations to MPC gates is defined inside the analysis, and there is no need to explicitly perform this transformation.) We analyze MPC-source and generate the linear program. We then pass the linear program to MATLAB and solve it using its built-in LP solver.

7.2 Our Experiments

OPA is parameterized by the cost model and its optimality is with respect to the underlying costs. We detail how we obtained cost for our experiments in Section §7.3. Using our toolchain with these costs we run the following experiments on a Core i6-6500 3.2 GHz computer with 16GB of RAM: For each benchmark, and each pair of protocols from $\{\pi^{\mathtt{A}}, \pi^{\mathtt{B}}, \pi^{\mathtt{Y}}\}$ (i.e. for each of $\{\pi^{\mathtt{A}}, \pi^{\mathtt{B}}\}$, $\{\pi^{\mathtt{A}}, \pi^{\mathtt{Y}}\}$ and $\{\pi^{\mathtt{B}}, \pi^{\mathtt{Y}}\}$), we plugged in our corresponding costs to derive the linear program. We used the solutions of the corresponding linear programs to obtain the optimal 2-out-of-3 protocol assignment by keeping the one with the overall minimum value for the objective function. The results of the experiment are summarized in Table 1.

7.3 Calculating Costs

Calculating accurate costs is of high importance for the usability of any protocol assignment tool. As discussed in §3.5, one can instantiate the cost model with different cost values depending on the setting. Following the trend in the hybrid MPC literature [DSZ15; MR18; Büs+18] we focused on *running time* in our experiments. Ideally, we would reuse cost tables from existing works for the most accurate comparison. Unfortunately, not all costs are reported, and the actual code that runs the experiment is not released at the time of writing. Therefore, we use the following methodology to calculate runtime costs of different instructions and share conversions.

To compute costs for an operation `OP` in the unamortized setting, we use a circuit with two inputs `a, b` from Alice and Bob, a gate `OP` that operates on `a, b` and a reconstruction gate that reconstructs to both parties. To facilitate comparison with ABY [DSZ15], we obtain the circuits by use of the public interface of ABY [DSZ15] without modifying the internal code, i.e., we use ABY's circuit creation mechanism in a black-box manner. We run this circuit 1000 times and average the total time reported by ABY [DSZ15]. For n parallel/amortized operations, we have a circuit with n copies of each gate (as described above, in the unamortized setting), making n `OP` gates execute in parallel. Observe that the unamortized setting is exactly $n = 1$. We create and run experiments for $n = \{1, 2, 5, 10, 25, 50, 100, 200, 300, 500, 800\}$.

An important factor that affects the run time of MPC is the communication network. Therefore, one usually investigates two common scenarios: execution over a Local Area Network (LAN) vs. over Wide Area Network (WAN). There are two types of experiments one can do to estimate the effect of the network, namely execute the protocol over a real LAN or WAN [DSZ15], or use a network simulator [MR18; Büs+18]. As our goal is mainly to demonstrate our toolbox and compare to existing results, we used the latter method. We note that in either case, existing benchmarks demonstrate that although the network type affects the absolute running time of the protocols, in almost all cases it does not affect the actual optimal assignment. This trend is confirmed in our simulation experiments.

Following the methodology used in [MR18; Büs+18] we used Linux's Traffic Control `tc` to simulate the network. We used the same parameters as in [MR18]: LAN (i.e., bandwidth=10gbps, burst=250mbps,

Table 2. Instruction cost, in micro-seconds. Averaged over 100 executions except when $n = 1$ where it is averaged over 1000 executions. (32bit)

Inst		**n (Simulated LAN)**			**n (Simulated WAN)**		
		1	100	500	1	100	500
ADD	B	2083	151	134	2088	1706	1484
	Y	1476	77	66	1473	468	801
	A	908	9	3	897	10	3
AND	B	1372	34	30	1369	43	276
	Y	1462	50	50	1461	504	937
EQ	B	1838	37	33	1863	50	249
	Y	1457	52	49	1454	394	772
GE	B	2134	92	87	2145	907	999
	Y	1487	74	71	1485	642	1111
GT	B	2026	69	67	2020	577	855
	Y	1463	54	51	1466	649	990
LE	B	2018	72	67	2016	512	709
	Y	1468	54	52	1467	405	739
LT	B	2136	93	88	2141	1094	1020
	Y	1479	74	72	1470	1094	1083
MUL	B	5831	1992	1963	5811	12212	12117
	Y	2812	1139	1114	2118	13553	11867
	A	3057	20	17	3136	40	197
MUX	B	1405	26	24	1409	37	24
	Y	1474	61	59	1459	433	763
NE	B	1855	38	32	1851	51	271
	Y	1452	52	49	1465	482	718
OR	B	1381	44	41	1393	53	259
	Y	1463	61	58	1459	526	776
SHL	B	2511	370	369	2493	3235	3336
	Y	1797	260	249	1776	4413	3807
SHR	B	2521	379	375	2492	3762	3517
	Y	1775	258	253	1785	3811	3680
SUB	B	4449	72	63	4513	101	330
	Y	1490	70	67	1477	635	766
	A	910	4	3	915	10	3
XOR	B	925	17	17	923	23	16
	Y	1398	40	39	1394	214	537
A2B		1772	138	130	1758	1815	1520
A2Y		1690	134	129	1705	1250	1753
B2A		1439	39	37	1444	85	440
B2Y		1536	65	60	1519	527	893
Y2A		1967	56	52	1977	342	710
Y2B		1463	44	42	1460	221	583

latency=500us) vs. WAN (i.e., bandwidth=40mbps, burst=1mbps, latency=40ms). The target machine for cost calculation is a virtual machine with a single 3.2GHz core with 4GB of RAM. The collected results are shown in Table 2.

The above cost table demonstrates the standard cost trends reported in the literature: Amortized operations are less costly (per operation) than unamortized one, the Yao-based protocol performs better than Boolean (GMW) in most cases, and Arithmetic performs better when amortized. However, a closer look reveals a cacophony which calls for a re-examination of how cost are computed throughout the mixed protocol selection literature: The costs for unamortized, or slightly-amortized operations are similar in (simulated) LAN and WAN. This is not surprising if one takes into account that: (1) the simulated WAN is effectively a less powerful LAN (i.e., issues resulting from using different routing protocols do not appear in such simulations); and (2) since there is no other traffic flowing thought the simulated network, the two networks perform very similarly in low-load scenarios.

The first issue can be mitigated by running the experiments over actual networks, but solving the second issue is tricky: One might be tempted to decrease the capacity (or, equivalently, increase the flow) of the simulated network. This will indeed make saturation take effect even with low traffic and create a cost difference in WAN vs LAN. However, this may still not capture the actual cost of operations, since this cost depends on what protocol traffic is circulated, e.g., if an operation OP2 follows a parallel batch of a communication-intensive operation OP1, then it might be better to compute OP2 with a protocol which is more computation-intensive but less-communication intensive. Although it does not affect our theory, we view cost measurement as an important open problem for this line of work.

Importantly, the trends observed in [DSZ15; Büs+18] that dominate protocol assignment, are present in our cost measurements as well. As a result, our toolchain computes consistent protocol assignments with previous works, as discussed in the following section.

7.4 Evaluation of the Implementation

In this section we describe the results of running our OPA solver. We run the solver on benchmarks from HyCC[Büs+18] (`https://gitlab.com/securityengineering/HyCC`) [10] and ABY[DSZ15], as well as a couple of new ones that we constructed for these experiments. We compare the outcome of our solver to the assignments proposed in HyCC[Büs+18] and ABY[DSZ15]. The results are summarized in Table 1.

In the following, we discuss the outcome of each of the benchmarks in Table 3 and, wherever feasible, confirm that our OPA solver demonstrates the expected behavior.

1) GCD This is the running example from this paper. Alice and Bob compute the GCD of their inputs. This protocol entails no parallelization (i.e., no amortization). Since cost of sequential operations in π^{Y} is the least, the IP outputs that assignment.

2) Biometric Matching A server holds a database S containing m n-dimensional tuples, and the client holds an n-dimensional query C. The parties compute the tuple in the database with the minimal euclidean distance to the query C (here $m = 512, n = 4$). This is a standard MPC benchmark whose assignment is well known. It has a two pass structure. In the first pass, arithmetic operations are highly parallelized. Therefore, the first pass is assigned the arithmetic protocol π^{A}. The second pass computes the minimum and uses (unamortized) comparison (GE) and multiplexing (MUX) operations. Both of these cost less in π^{Y} than in π^{B}. Therefore, the second pass is assigned π^{Y}. There is a single array that contains the output of the first pass, therefore, a single conversion happens before the second pass. Our assignment is the same as the well known assignment.

3) Modular Exponentiation Two parties come together to compute $base^{exp} \mod m$ ($base, exp$ and mod are all 32 bit unsigned integers) where one party holds $base$ and the other party holds exp. This protocol accepts no parallelization either and is assigned π^{Y} for the same reason as GCD. In the (simulated) WAN setting,

[10] We have translated all HyCC's publicly available benchmarks, except, due to time constraints, Gauss.

Table 3. Assignments Comparison with HyCC[Büs+18] and ABY[DSZ15]. For easier notation, we use A, B, and Y instead of π^{A}, π^{B}, and π^{Y}.

	Simulated LAN			**Simulated WAN**		
Benchmark	OPA Solver	HyCC	ABY	OPA Solver	HyCC‡	ABY
GCD	Y	—	—	Y	—	—
Biometric Matching	A+Y(4,3,1)*	A+Y	A+Y	A+Y(4,3,1)*	A+Y	A+Y
Modular Exponentiation	Y	—	A+B+Y	Y	—	Y
Private Set Intersection (PSI)	B	—	B†	B	—	B+Y†
Histogram	A+Y(3,21,1)*	—	—	A+Y(3,21,1)*	—	–
MiniONN (MNIST)	A+Y(65,49,7)*	A+Y	—	A+Y(65,49,7)*	A+Y	–
k-means	B+Y(2,34,2)*	A+Y	—	B+Y(2,34,2)*	A+Y	–
DB-Merge (500 + 500)	A+Y(5,21,4)*	A+Y	—	A+Y(5,21,4)*	A+Y	–
DB-Join (50 x 50)	A+Y(6,27,2)*	A+Y	—	A+Y(6,27,2)*	A+Y	–
DB-Join (25 x 200)	A+Y(6,37,3)*	A+Y	—	A+Y(6,37,3)*	A+Y	–
Cryptonets (Square)	A+Y(24,15,1)*	A	—	A+Y(24,15,1)*	A	–

— Assignment not provided.
† ABY[DSZ15] does not specify which implementation of PSI it uses, therefore comparison is not meaningful.
‡ We use the assignment in HyCC[Büs+18] that yields minimum total time (setup + online).
* The first value in the triplet is # of operations in A or B depending on the assignment, A+Y or B+Y respectively. The second value is # of operations in Y, the third value is # of conversions.

our assignment is the same as ABY's [DSZ15]. In the (simulated) LAN seting, ABY [DSZ15] assigns a combination of all three protocols using a faster MUX—whose implementation is not publicly available. Our assignment, π^{Y}, which we computed using the standard implementation of MUX is their second best.

4) Private Set Intersection (PSI) A server holds set $S1$, a client holds set $S2$ (here sizes of $S1$ and $S2$ are 1024 and 32 respectively, elements are 32-bit unsigned integers). We use the straightforward $O(n^2)$ protocol. It is completely parallelizable and relies on NE and MUX operations. Looking at the cost tables, the amortized NE and MUX are cheaper with π^{B}, therefore the π^{B} assignment.

5) Histogram This is a benchmark that we adapted from the PUMA benchmark suite of MapReduce programs. Parties jointly hold a movie ratings database of n reviewers and m movies (here $n = 100, m = 100$, and all elements are unsigned integers). Together, they compute a histogram of average ratings of the reviewers. It has one loop with enough parallelization to justify a π^{A} assignment, hence the optimal assignment mixes π^{A} and π^{Y}.

6) MiniONN [Liu+17] (MNIST) and Cryptonets [Gil+16] These are Machine Learning benchmarks. We translated them from HyCC [Büs+18]'s public code and ran them through our toolchain. MNIST is the largest benchmark in terms of lines of code, and the most complex one. Several loops with arithmetic operations are parallelizable, and those loops are assigned π^{A}; all other operations are assigned π^{Y}. This makes the summary assignment mixing π^{A} and π^{Y}, the same as reported by HyCC [Büs+18]. In Cryptonets (RELU function being square), although there are only arithmetic operations, some of them are inside non-parallel loops. Because our unamortized π^{Y}-costs are less than unamortized π^{A} (which is standard in the WAN setting), arithmetic operations in the non-parallel loops are assigned π^{Y}. This makes the full assignment a mix of π^{A} and π^{Y}. By comparison, the assignment from HyCC [Büs+18] uses π^{A} only; although HyCC does not report the relevant costs, we believe that the reason for this assignment is that their cost of arithmetic operations in π^{A} is less than in π^{Y}.

7) k-means This is a clustering algorithm and a data mining benchmark. We took it from HyCC [Büs+18]'s public code and ran it through our toolchain. We did not detect parallelizable loops, which explains the lack of assignments to $\pi^{\mathtt{A}}$. There is an OR operation (in the implementation of integer division) whose result is accumulated for subsequent operations. This gets an assignment of $\pi^{\mathtt{B}}$. Our overall assignment is then a mix of $\pi^{\mathtt{B}}$ and $\pi^{\mathtt{Y}}$. HyCC's assignment is a mix of $\pi^{\mathtt{A}}$ and $\pi^{\mathtt{Y}}$. The reason we do not detect any $\pi^{\mathtt{A}}$ assignments is that we analyze the standard version, and we do not detect parallelization. HyCC analyzes a parallelizable version, hence the $\pi^{\mathtt{A}}$ assignment to arithmetic operations.

8) DB-Merge (500 + 500), DB-Join (50x50) and DB-Join (25 x 200) These are data analytics benchmarks, also taken from HyCC [Büs+18]. All of these contain some arithmetic operations inside parallelizable loops. Therefore those operations are assigned $\pi^{\mathtt{A}}$. The overall assignment that optimizes total time in all three cases is a mix of $\pi^{\mathtt{A}}$ and $\pi^{\mathtt{Y}}$, just as in HyCC.

8 Conclusions

We revisit the problem of optimal protocol assignment (OPA) for hybrid MPC which was conjectured to be NP-hard. We prove that, modulo scheduling/parallelization, for the special case of two protocols, the problem can in fact be solved in polynomial time. Our analysis is based on a framework we propose which combines ideas and techniques from program analysis and MPC. We implemented our OPA solver and tested it using simulated costs in a wide set of known benchmarks demonstrating its efficiency and quality. Our treatment points to several open problems in programming language, MPC, and networks.

References

[Aho+06] Alfred V. Aho et al. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[Bea92] Donald Beaver. "Efficient Multiparty Protocols Using Circuit Randomization." In: *CRYPTO'91*. Ed. by Joan Feigenbaum. Vol. 576. LNCS. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 1992, pp. 420–432. DOI: 10.1007/3-540-46766-1_34.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract)." In: *20th ACM STOC*. Chicago, IL, USA: ACM Press, May 1988, pp. 1–10. DOI: 10.1145/62212.62213.

[BK15] Niklas Büscher and Stefan Katzenbeisser. "Faster Secure Computation through Automatic Parallelization." In: *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. 2015, pp. 531–546. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/buescher.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. "Sharemind: A Framework for Fast Privacy-Preserving Computations." In: *ESORICS 2008*. Ed. by Sushil Jajodia and Javier López. Vol. 5283. LNCS. Málaga, Spain: Springer, Heidelberg, Germany, Oct. 2008, pp. 192–206. DOI: 10.1007/978-3-540-88313-5_13.

[BNP08] A. Ben-David, N. Nisan, and B. Pinkas. "FairplayMP: a system for secure multi-party computation." In: *Proc. 15th ACM Conf. Comput. and Commun. Security (CCS)*. Alexandria, VA, USA: ACM, 2008, pp. 257–266.

[Büs+18] Niklas Büscher et al. "HyCC: Compilation of Hybrid Protocols for Practical Secure Computation." In: *ACM CCS 18*. Ed. by David Lie et al. Toronto, ON, Canada: ACM Press, Oct. 2018, pp. 847–861. DOI: 10.1145/3243734.3243786.

[Cam65] Paul Camion. "Characterization of Totally Unimodular Matrices." In: *Proceedings of the American Mathematical Society* 16.5 (1965), pp. 1068–1073.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. "Multiparty Unconditionally Secure Protocols (Extended Abstract)." In: *20th ACM STOC*. Chicago, IL, USA: ACM Press, May 1988, pp. 11–19. DOI: 10.1145/62212.62214.

[Cha+17] Nishanth Chandran et al. "EzPC: Programmable, Efficient, and Scalable Secure Two-Party Computation." In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 1109.

[Cho+07] Stephen Chong et al. "Secure Web Applications via Automatic Partitioning." In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 31–44. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294265. URL: http://doi.acm.org/10.1145/1294261.1294265.

[Cho+13] Ashish Choudhury et al. "Between a Rock and a Hard Place: Interpolating between MPC and FHE." In: *ASIACRYPT 2013, Part II*. Ed. by Kazue Sako and Palash Sarkar. Vol. 8270. LNCS. Bengalore, India: Springer, Heidelberg, Germany, Dec. 2013, pp. 221–240. DOI: 10.1007/978-3-642-42045-0_12.

[Cyt+91] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: http://doi.acm.org/10.1145/115372.115320.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. "ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation." In: *NDSS 2015*. San Diego, CA, USA: The Internet Society, Feb. 2015.

[Fra+14] Martin Franz et al. "CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations." In: *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*. 2014, pp. 244–249. DOI: 10.1007/978-3-642-54807-9_15. URL: https://doi.org/10.1007/978-3-642-54807-9%5C_15.

[Gil+16] Ran Gilad-Bachrach et al. "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy." In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2016, pp. 201–210. URL: http://jmlr.org/proceedings/papers/v48/gilad-bachrach16.html.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority." In: *19th ACM STOC*. Ed. by Alfred Aho. New York City, NY, USA: ACM Press, May 1987, pp. 218–229. DOI: 10.1145/28395.28420.

[Hen+10] Wilko Henecka et al. "TASTY: tool for automating secure two-party computations." In: *ACM CCS 10*. Ed. by Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov. Chicago, Illinois, USA: ACM Press, Oct. 2010, pp. 451–462. DOI: 10.1145/1866307.1866358.

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. "Semantical interprocedural parallelization: an overview of the PIPS project." In: *Proceedings of the 5th international conference on Supercomputing, ICS 1991, Cologne, Germany, June 17-21, 1991*. 1991, pp. 244–251. DOI: 10.1145/109025.109086. URL: https://doi.org/10.1145/109025.109086.

[IMZ19] Muhammad Ishaq, Ana Milanova, and Vassilis Zikas. "Efficient MPC via Program Analysis: A Framework for Efficient Optimal Mixing." In: *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom*. 2019. ISBN: 978-1-4503-6747-9/19/11. DOI: 10.1145/800057.808695.

[Kar84] N. Karmarkar. "A New Polynomial-time Algorithm for Linear Programming." In: *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*. STOC '84. New York, NY, USA: ACM, 1984, pp. 302–311. ISBN: 0-89791-133-4. DOI: 10.1145/800057.808695. URL: http://doi.acm.org/10.1145/800057.808695.

[Kha80] Leonid G. Khachiyan. "A Polynomial-Time Algorithm for Solving Linear Programs." In: *Mathematics of Operations Research* 5.1 (Feb. 1980). ISSN: 0364-765X. DOI: 10.1287/moor.5.1.iv.

[KSS13] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. "A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design." In: *Journal of Computer Security* 21.2 (2013), pp. 283–315. URL: http://dblp.uni-trier.de/db/journals/jcs/jcs21.html#KolesnikovS013.

[KSS14] Florian Kerschbaum, Thomas Schneider, and Axel Schröpfer. "Automatic Protocol Selection in Secure Two-Party Computations." In: *ACNS 14*. Ed. by Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay. Vol. 8479. LNCS. Lausanne, Switzerland: Springer, Heidelberg, Germany, June 2014, pp. 566–584. DOI: 10.1007/978-3-319-07536-5_33.

[Liu+17] Jian Liu et al. “Oblivious Neural Network Predictions via MiniONN Transformations.” In: *ACM CCS 17*. Ed. by Bhavani M. Thuraisingham et al. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 619–631. DOI: 10.1145/3133956.3134056.

[MR18] Payman Mohassel and Peter Rindal. “ABY^3: A Mixed Protocol Framework for Machine Learning.” In: *ACM CCS 18*. Ed. by David Lie et al. Toronto, ON, Canada: ACM Press, Oct. 2018, pp. 35–52. DOI: 10.1145/3243734.3243760.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer International Publishing, Inc., 2014. ISBN: 3319105418 9783319105413.

[NNH10] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010. ISBN: 3642084745, 9783642084744.

[Pat+16] Erman Pattuk et al. “CheapSMC: A Framework to Minimize SMC Cost in Cloud.” In: *CoRR* abs/1605.00300 (2016). arXiv: 1605.00300. URL: http://arxiv.org/abs/1605.00300.

[Sch03] Alexander Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. 1st. Springer-Verlag Berlin Heidlberg, 2003. ISBN: 978-3-540-44389-6, 0937-5511.

[Sco15] Michael L. Scott. *Programming Language Pragmatics (2Nd Edition)*. Morgan Kaufmann, 2015. ISBN: 0124104096.

[SK11] Axel Schröpfer and Florian Kerschbaum. “Forecasting Run-Times of Secure Two-Party Computation.” In: *Eighth International Conference on Quantitative Evaluation of Systems, QEST 2011, Aachen, Germany, 5-8 September, 2011*. 2011, pp. 181–190. DOI: 10.1109/QEST.2011.33. URL: https://doi.org/10.1109/QEST.2011.33.

[SKM11] Axel Schröpfer, Florian Kerschbaum, and Günter Müller. “L1 - An Intermediate Language for Mixed-Protocol Secure Computation.” In: *COMPSAC*. IEEE Computer Society, 2011, pp. 298–307. ISBN: 978-0-7695-4439-7. URL: http://dblp.uni-trier.de/db/conf/compsac/compsac2011.html#SchropferKM11.

[SR18] Nigel Smart and Dragos Rotaru. *SCALE-MAMBA*. 2018. URL: https://github.com/KULeuven-COSIC/SCALE-MAMBA (visited on 11/22/2018).

[Val+99] Raja Vallée-Rai et al. “Soot - a Java Bytecode Optimization Framework.” In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON ’99. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13–. URL: http://dl.acm.org/citation.cfm?id=781995.782008.

[Wil+94] Robert P. Wilson et al. “SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers.” In: *SIGPLAN Notices* 29.12 (1994), pp. 31–37. DOI: 10.1145/193209.193217. URL: https://doi.org/10.1145/193209.193217.

[Yao82] Andrew Chi-Chih Yao. “Protocols for Secure Computations (Extended Abstract).” In: *23rd FOCS*. Chicago, Illinois: IEEE Computer Society Press, Nov. 1982, pp. 160–164. DOI: 10.1109/SFCS.1982.38.

A Notation

In this appendix we cover notation and terminology that is used through out the paper.

A.1 General Terminology

- *(IMP-)source code*: This is the starting point of our compiler. It is standard programming language code for an imperative language such as IMP. We denote it by S. All loops have a known upper bound on their iterations.
- *MPC-source code*: The output of our compiler on some source code S. We denote the compiler by $C_{\text{MPC}}(\cdot)$. The compiler removes if-statements and ϕ-nodes, and adds MUX-statements in their place. MPC-source contains for-loops with known bounds.
- *Block B of MPC-source*: Sequence of assignment statements or blocks (in case of for-loop nesting) enclosed in a for-loop.

- *(IMP-)SSA-code*: this is the output of SSA on some source-code S. We will denote it as $C_{\text{SSA}}(S)$. This is an intermediate representation between (IMP-)source and MPC-source.
- *Linearized-code :* $\mathsf{Linear}(S)$: This is the linearization of some MPC-source $C_{\text{MPC}}(\cdot)$. It contains no loops, only straight-line code of assignment statments. The corresponding CFG of this would be simply a straight line.
- We refer to statement in $\mathsf{Linear}(S)$ as *simple statements* and denote them as $\mathtt{st}$. Since the corresponding CFG is a line we often refer to simple statements as *nodes in (the CFG of)* $\mathsf{Linear}(S)$.

A.2 Costs Model

Simple Model

- $St = \{\mathtt{st}_1, \ldots, \mathtt{st}_\ell\}$ denotes the ordered set of statements in $\mathsf{Linear}(S)$
- $\Pi = \{\pi_1, \ldots, \pi_m\}$ denotes (a set of) multi-party protocols and $\Sigma = \{\sigma_1, \ldots, \sigma_q\}$ denotes (a set) of secret sharing schemes (in typical scenarios such as [DSZ15; MR18; Cha+17; Büs+18] $q = m$.)
- For each $(i, j) \in [\ell] \times [m]$, the triple $(\mathtt{st}_i, \pi_j, c^{\pi_j}_{\mathtt{st}_i}) \in B \times \Pi \times \mathbb{Z}_{\geq 0}$, where intuitively, $c^{\pi_j}_{\mathtt{st}_i}$ corresponds to the cost of emulating in a flow statement $\mathtt{st}_i$ with protocol π_j.
- For each $(i, j) \in \Sigma^2$: the triple $(\sigma_i, \sigma_j, c^{\sigma_i 2 \sigma_j}) \in \Sigma \times \Sigma \times \mathbb{Z}_{\geq 0}$, where intuitively, $c^{\sigma_i 2 \sigma_j}$ is the cost of securely converting a sharing according to scheme σ_i into a sharing according to σ_j.
- Whenever the sequence St, and set Π are clear from the context we use $c^{\pi_j}_{\mathtt{st}_i}$'s and $c^{\sigma_i 2 \sigma_j}$ instead of the setup of triples. Furthermore, in all existing works on protocol mixing—including ours—each protocol π_i is associated with a single sharing scheme σ_i; in such cases, in slight abuse of notation, we denote the conversion cost from σ_i to σ_j as $c^{\pi_i 2 \pi_j}$ (instead of $c^{\sigma_i 2 \sigma_j}$). In fact, to further simplify our notation and consistently with the ABY notation, for the three ABY protocols $\pi^{\mathtt{A}}, \pi^{\mathtt{B}}$, and $\pi^{\mathtt{Y}}$, and for $X, Z \in \{\mathtt{A}, \mathtt{B}, \mathtt{Y}\}$ we use c^{X2Z} to denote the conversion cost $c^{\pi^X 2 \pi^Z}$ from the sharing corresponding to π^X (which we refer to as Sharing X) to the sharing corresponding π^Z (which we refer to as Sharing Z).

Amortized Model

- The triplet $(\mathtt{st}_i, \pi_j, f_{c^{\pi_j}_{\mathtt{st}_i}}(\cdot))$, where $f_{c^{\pi_j}_{\mathtt{st}_i}} : \mathbb{N} \to \mathbb{Z}_{\geq 0}$ denotes the amortized execution cost function, which on input $\ell \in \mathbb{N}$ outputs the amortized cost $f_{c^{\pi_j}_{\mathtt{st}_i}}(\ell)$ of computing ℓ parallel copies of $\mathtt{st}_i$ with protocol π_j.
- The triplet $(\sigma_i, \sigma_j, f_{c^{\sigma_i 2 \sigma_j}}(\cdot))$, where $f_{c^{\sigma_i 2 \sigma_j}} : \mathbb{N} \to \mathbb{Z}_{\geq 0}$ denotes the amortized conversion cost function, which on input $\ell \in \mathbb{N}$ outputs the amortized cost $f_{c^{\sigma_i 2 \sigma_j}}(\ell)$ of converting ℓ sharings according to σ_i into sharings according to σ_j.
- For brevity, for $X, Z \in \{\mathtt{A}, \mathtt{B}, \mathtt{Y}\}$ we use $f_{c^{X2Z}}$ to denote the function $f_{c^{\pi^X 2 \pi^Z}}$ from the sharing corresponding to π^X to the sharing corresponding π^Z. The costs of the simple model corresponds to the output of the above functions on input $\ell = 1$.

OPA for Linearized MPC

- PA is a sequence of pairs of the type $(\mathtt{st}_1, \pi_1), \ldots, (\mathtt{st}_{|St|}, \pi_{|St|})$ where $(\mathtt{st}_i, \pi_j) \in \mathsf{PA}$ means that statement $\mathtt{st}_i$ is assigned protocol π_j.

A.3 Solving The Linearized OPA

- c^A_n is the cost to run node $n \in C_{\text{MPC}}(S)$
- c^Y_n is the cost to run node n using $\pi^{\mathtt{Y}}$.
- c^{A2Y} is the cost to run A2Y conversion.
- c^{Y2A} is the cost to run Y2A conversion.

Variables and Constraints

– $(d, u) \supseteq (d, u')$ denotes that (d, u) *subsumes* (d, u') i.e. all paths from d to u' go through $min\_cut(d, u)$.

From $IP_{\mathsf{Linear}(S)}$ to $IP_{C_{\text{MPC}}(S)}$

– $\alpha : \mathsf{Linear}(S) \rightarrow C_{\text{MPC}}(S)$ denotes the "abstraction" function i.e. provides mapping from $\mathsf{Linear}(S)$ to $C_{\text{MPC}}(S)$.
– $\gamma : (C_{\text{MPC}}(S) \times C_{\text{MPC}}(S)) \rightarrow 2^{\mathsf{Linear}(S)}$ denotes the "concretization" function i.e. provides mapping from $C_{\text{MPC}}(S)$ to $\mathsf{Linear}(S)$.

B Preliminaries

B.1 Program Analysis

We next discuss concepts that are standard building blocks of static analysis and are necessary background for our results. We assume minimal familiarity with program analysis, and refer an interested reader to [Aho+06].

Basic Block (BB) A basic block (BB) is a straight-line sequence of instructions, defined by the compiler. The set of basic blocks that may execute before a given basic block are called its *predecessors*. Similarly, the set of blocks that may execute after a given block are called its *successors*.

Control Flow Graph (CFG) A control flow graph (CFG) is a directed graph that represents all possible control flow paths in a program. The nodes in the CFG are basic blocks, and the edges model flow of control between basic blocks. There is an edge from a basic block to each of its successors. It is also common to consider each statement in a basic block as a separate node with an outgoing edge to the statement/node immediately following within the basic block.

Reaching Definitions (RDs) Reaching definitions is a classical data-flow analysis technique [Aho+06; NNH10]. It computes *def-use chains* (d, u), where d is a definition of a variable x: e.g., $\mathsf{x} = \mathsf{y} + \mathsf{z}$, and u is a use of x: e.g., $\mathsf{z} = \mathsf{x} * \mathsf{y}$, or $\mathsf{x} > \mathsf{y}$. In the classical sense, reaching definitions is defined over a CFG, where d and u are statements/nodes in the graph. A def-use chain (d, u) entails that there is a path from d to u in the CFG that is free of a definition of x, or in other words, the definition of x at d *may* reach the use of x at u.

Reasoning about dependencies like def-use chains can be greatly simplified by an appropriate *intermediate representation* (IR). Now, we describe an intermediate representation (IR) called *Static Single Assignment* (SSA) form. This is a standard IR in compilers and benefits static analysis by immediately exposing def-use dependencies. The standard algorithm to translate a program into SSA form is due to Cytron et al. [Cyt+91].

Static Single Assignment (SSA) form SSA form entails that each variable in the program is assigned *exactly once*. If the source code has multiple definitions of the same variable, the variable is split into multiple versions for each definition. Consider, for instance, the code fragment in Figure 5(a). Without SSA, a compiler needs to construct def-use chains to reason that the first definition of x is not used and is, therefore, dead code. Now consider the same code fragment in SSA form in Figure 5(b). It is immediately obvious that variable x1 has no uses. Moreover, it is also obvious—because all variables are assigned only once—that y is only a copy of x2. Therefore, in any uses of y, y can be replaced with x2 without changing the input program behavior. Furthermore, x2 is a constant with value 2, and consequently z is a constant too, with value 200. The final SSA-program will just use the constant value 200 and will eliminate the variables in the original program in Figure 5(a).

A natural question is, if SSA form allows variable assignment only once, how does it determine which variable to use when multiple control flow paths merge into a single node e.g. the if-else in figure 6(a). This is taken care of in SSA by so-called *phi (ϕ) nodes.*

```
x = 1;
x = 2;
y = x;
z = y * 100;
```
(a)

```
x1 = 1;
x2 = 2;
y = x2;
z = y * 100;
```
(b)

Figure 5. A simple source program and its SSA form. x_1 = 1 is dead code and also, y is just a copy of x_2. y is a constant, and z is a constant with value of 200.

```
if (flag) {
    x1 = 1;
}
else {
    x2 = 2;
}
x3 = x?; // Is x3 x1 or x2?
```
(a)

```
if (flag) {
    x1 = 1;
}
else {
    x2 = 2;
}
x3 = φ(x1, x2);
```
(b)

Figure 6. A program and its SSA form. We assume that the first argument of a ϕ node (x1 in our case) carries the value along the then-arm of the if-statement, and the second argument (x2 in our case) carries the value along the else-arm.

Phi (ϕ) Nodes ϕ-nodes follow immediately after control-flow from two or more paths joins (merges) into a single node. They have the form $\mathsf{x3} = \phi(\mathsf{x1}, \mathsf{x2})$, where x3, is a new version of the variable, and $\phi(\mathsf{x1}, \mathsf{x2})$, contains the versions of the variable along the different paths. The ϕ-node entails that x's value at this point comes from either the then-arm (x1) or the else-arm (x2) depending on what path control flow took to arrive at the merge node. Figure 6(b) shows the SSA form (including a ϕ-node) corresponding to code in Figure 6(a).

IMP Imperative Language Recall that one of our goals in this work is to define *MPC-source*, the input IR for MPC compilers/optimizers. Towards this goal, we start from a standard representation of program syntax. The standard representation in the *functional* programming languages literature uses lambda calculus. However, MPC programs live in the *imperative* world. Therefore, we choose a standard minimal representation of an imperative language, IMP. IMP (cf. [NK14, ch. 7]) is a simple programming language in which a statement can either be an 1) assignment to an expression where expression can be a constant, a variable or an operation between two variables, 2) an if-then-else conditional or 3) a while loop.

C Program Analysis of MPC Source

C.1 Program Syntax

```
// Computes val%mod
int rem = val;
while (rem ≥ mod)
rem = rem - mod;
return rem;
```
(a) Remainder

```
int x = a;
int y = b;
while (y != 0) {
  r = rem(x,y);
  x = y;
  y = r;
}
return x;
```
(b) GCD

Figure 7. Standard algorithms for Remainder and GCD

We assume an IMP-like source syntax [NK14]. The IMP syntax models an imperative language, such as FORTRAN, C, or Java, and our results apply to any of these languages. We impose the following standard restrictions necessary to accommodate MPC: there is no recursion, and all loop-bounds are statically known. The IMP source is translated into Static Single Assignment (SSA) using standard techniques [Cyt+91]. Fig. 8 abstracts the SSA syntax corresponding to IMP-like sourecode. Note that this is standard SSA, however, to make it explicit that it corresponds to IMP-source, in the following we will refer to it as *IMP-SSA*. This is the syntax of our intermediate representation. (Note that this is also the representation that Shimple [Val+99] produces when executed on IMP-source code.)

For readers unfamiliar with SSA we discuss the basic features of the IMP-SSA representations. The IMP-SSA program is a sequence of statements, where each statement is either (1) a copy propagation assignment, e.g., x = y, (2) a three-address assignment, e.g., x = y+z (3) a for-loop statement, or (4) an if-then-else statement. An if-statement is immediately followed by one or more ϕ-nodes, as is standard SSA form. (One may need more than one ϕ nodes when more than one variables are assigned along one or both branches of the if-then-else.) In the running example in Fig. 1(b) lines 11-24 show the IMP-SSA translation of method rem, where rem is inlined into gcd. As it is standard in SSA, each assignment yields a new version of the variable on the left-hand-side, e.g., we have rem2, rem3, rem4. Control flow merge at the end of the if-statement entails ϕ-nodes. In our running example, rem5 = ϕ(rem4,rem3) at line 22 in Fig. 1(b) entails that if control took the then-arm of the if-statement, rem has the value of rem4, otherwise, rem has the value of rem3. We assume that the first argument of a ϕ node carries the value along the then-arm of the if-statement, and the second argument carries the value along the else-arm.

C.2 Translation to MPC-source

We next discuss how our intermediate representation of IMP-SSA is translated to the representation that we use for defining our compact integer program, which we call *MPC-source*.

s	$::= s; s$	
	$\mid \mathsf{x} = \mathsf{y}$	
	$\mid \mathsf{x} = \mathsf{y}\ \mathsf{aop}\ \mathsf{z}$	
	$\mid \mathsf{a}[i] = \mathsf{x}$	
	$\mid \mathsf{x} = \mathsf{a}[i]$	
	$\mid \mathsf{for}\ (i = 0; i \leq n; i{+}{+})\ \{\ s\ \}$	
	$\mid \mathsf{if}\ (\mathsf{x}\ \mathsf{bop}\ \mathsf{y})\ \{\ s\ \}\ \mathsf{else}\ \{\ s\ \}\ \mathsf{z} = \phi(\mathsf{z}_1, \mathsf{z}_2)$	*statement*
aop	$::= + \mid - \mid * \mid /$	*arithmetic operator*
bop	$::= == \mid\ != \mid < \mid \leq$	*comparison operator*

Figure 8. IMP-SSA syntax. s represents a sequence of statements. x, y, and z denote variables, including constants, local variables, and parameters that hold shares. i and n denote variables in plain text. Note that each if-then-else statement is immediately followed by a ϕ-node, as is customary in SSA.

Fig. 2 defines an *attribute grammar* (also known as *syntax directed translation*) over the syntax in Fig. 8 that translates the IMP-SSA program into an MPC-source program. (An attribute grammar is a standard static analysis technique [Aho+06, Chapter 5], [Sco15, Chapter 4]; an attribute grammar is defined over the syntax of the program and performs semantic analysis or transformation.) In our case, this is a standard attribute grammar. The only interesting case arises at if-statements which are dealt with using standard MPC techniques: the MPC-source code for an if-statement is produced by appending the straight-line (MPC) code for the else-arm onto the straight-line (MPC) code for the then-arm, then adding the conditional, and the multiplexer to select the correct values. This is straight-forward given SSA: due to single assignment, variables used at the if-statement test are unmodified, and are referenced in the comparison expression (CMP) that precedes MUX, where the ϕ nodes capture exactly the arguments of MUX. [11] For example, consider the if-statement in lines 9-31 in Fig. 1(b). The ϕ nodes capture the values of x and y; if control took the then-arm, then x and y would be x2 and y2 respectively, otherwise x and y would be x1 and y1.

In our example, the resulting MPC-source program is shown in Fig. 1(c). We point out that MPC-source can be mapped one-to-one to standard straight-line MPC; the only difference is that when a block is repeated multiple times in straight-line MPC, it is replaced by a for-loop in MPC-source. Following standard MPC compilers methodology, e.g., [BNP08; Fra+14], the actual MPC program unrolls all loops, and loop induction variables become constants.

To make the above mapping explicit, we use *pseudo ϕ-nodes*. To better understand the use of these notes, let's focus on lines 5, 6 and 12 in Fig. 1(c) at the beginning of each one of the loops; these lines do not encapsulate an if-then-else construct. Instead, they select variable values—at the first iteration, the value comes from outside the loop, and at every subsequent iteration the value comes from the previous iteration of the loop. When translated into straight-line code, these lines disappear because corresponding values are directly used as inputs to the gates. To highlight that these lines are only here to enable loops, and, that these do not get translated into a MUX, we refer to them as *pseudo ϕ-nodes* in text and denote them with $?:$ instead of ϕ.

Looking ahead (cf. Section 4) the benefit of doing the analysis over MPC-source rather than straight-line code will be that there are significantly fewer variables in the resulting integer/linear program.

D Proofs for §5

D.1 Proof for Theorem 2

Before proving the theorem, let us first recall a results by Camion [Cam65] which we are using.

Definition 4. *A matrix M is said to be Eulerian, if the sum of the elements in each row of M is even, and the sum of the elements in each column of M is even.*

Theorem 5. *(Camion [Cam65]) Matrix M is totally unimodular if and only if for every square Eulerian submatrix of M $M^I_J : \sum_{i\in I, j\in J} M^j_i \equiv 0 \pmod 4$.*

In words, the above theorem states that a matrix is totally unimodular if and only if the sum of the elements of every square Eulerian submatrix is divisible by 4.

We return to our constraint matrix A. There are two kinds of rows in A:

1. Rows
$$\begin{array}{c} 1\,1\,0\,0\,0\,0\ldots \\ 0\,0\,1\,1\,0\,0\ldots \\ 0\,0\,0\,0\,1\,1\ldots \\ \ldots \end{array}$$
that reflect constraints $a^n + y^n \geq 1$. We use the term *first-kind rows* in the remainder of this section to describe these rows.
2. Rows
$$\begin{array}{c} -1\ \ 0\ \ \ldots 1\,0\ldots\ 1\ \ \ \ldots \\ 0\ \ -1\ldots 0\,1\ldots\ 0\ \ 1\ldots \\ \ldots \end{array}$$
reflect constraints $x^d_{e_1} + \cdots + x^d_{e_k} \geq a^u - a^d \equiv -a^d + a^u + x^d_{e_1} + \cdots + x^d_{e_k} \geq 0$. The first two non-zero entries in a row, a -1 and 1, reflect $-a^u$ and a^d; the remaining 1 entries reflect the x^d_e's. For each row of a and x^d_e-constraints (formula (2)), there is analogous row of y and z^d_e-constraints (formula (3)). We use the term *second-kind rows* in the remainder of this section.

[11] MUX is the multiplexer gate that is common in MPC compilers: on input of values (v_0, v_1) and a selection bit $b \in \{0,1\}$, it returns v_b. In our case b is result of the CMP and (v_0, v_1) are arguments of ϕ node.

We are now ready to prove the theorem. We first prove the following useful lemmas:

Lemma 1. *The representative edges of forward def-use chains are totally ordered by subsumption: $e_1 \supseteq e_2 \supseteq \cdots \supseteq e_k$.*

Proof. Suppose there exist two forward def-use chains (d, u) and (d, u') with representatives e and e', such that neither subsumes the other. Without loss of generality, we say that u precedes u'. By definition, e lies on the chain of forward edges from d to u and therefore, it dominates u' as well, meaning that e subsumes e'.

Lemma 2. *Let (d, u) be a backward def-use chain. We have*

1. *(d, u) does not subsume any other def-use chain*
2. *A forward chain (d, u') may subsume (d, u)*

Proof. As stated by Remark 2 in §3.6, each definition d gives rise to at most one backward def-use chain, where u is a pseudo-ϕ node in u's enclosing block. Also, as established in §3.6 (d, u)'s representative edge e is the backward edge of u's enclosing block. Assume then that (d, u) subsumes some forward chain (d, u'). There is an immediate contradiction because the chain of forward edges from u through d to u' does not pass through e. Therefore, (d, u) subsumes no other chain.

On the other hand, if $e' = min\_cut(d, u')$ is in d's enclosing loop block, then execution always passes through e', then e to reach u. Therefore, a forward (d, u') may subsume (d, u).

Proof. Suppose there exists an Eulerian submatrix of A, M, such that the sum of its elements is not divisible by 4. We prove the theorem for all Eulerian submatrices, and it follows for each square Eulerian submatrix.

Matrix M can be broken into two parts, submatrix M' which consists entirely of first-kind rows, and submatrix M'' which consists of second-kind rows. We have $M \equiv 2 \bmod 4$ only if one of the following is true: (1) $M' \equiv 2 \bmod 4$ and $M'' \equiv 0 \bmod 4$, or (2) $M' \equiv 0 \bmod 4$ and $M'' \equiv 2 \bmod 4$. (Here shortcut notation $M \equiv 2 \bmod 4$ denotes that the sum of the elements of M gives remainder 2 modulo 4.)

Consider case (1). If $M' \equiv 2 \bmod 4$, we must have an odd number of first-kind rows in M (Since each first-kind row has two 1 entries and an even number of rows would have given $M' \equiv 0 \bmod 4$). Consider the part consisting of a^n-entry 1's in M'. There is an odd number of these 1's. Since M is a Eulerian submatrix this means that each one of these 1's must be matched (i.e., evened out) in columns by entries from M''. Let aM'' be the submatrix which consists of a-rows, i.e., rows due to constraints: $-a^u + a^d + x^d_e + \cdots \geq 0$. The remainder of M'', which we denote by yM'' consists of rows due to constraints: $-y^u + y^d + x^d_e + \cdots \geq 0$. There is an odd number of columns in aM'' with odd sum each. (These must match the a^nentries.) However, the remaining columns of aM'' must have even sum each, since those columns are matched only within aM''. This implies that the sum of all elements of aM'' is odd (odd*odd + even). However, since M'' is Eulerian, meaning that each row in aM'' has even sum, it follows that the sum of all elements of aM'' is even, which leads to a contradiction. Therefore case (1) is impossible.

Consider case (2). We show that there is even number of rows in M'' with non-zero entries at x^d_e positions. If this is the case, then since each row has an even sum, the total sum of these rows is divisible by 4. There may be additional rows in M'' with entries at a^d and a^u positions, however since the a^d entry is 1 and the a^u entry is -1, these rows contribute 0 to the total sum of M''. By the same argument the sum of y (Yao) rows is divisible by 4, which entails that the sum of entries in M'' is divisible by 4, which contradicts the statement that $M'' \equiv 2 \bmod 4$.

We now argue that there is even number of rows in M'' with non-zero entries at x^d_e positions. Consider all x^d_e in M''. As argued earlier, the forward def-use chains are ordered by subsumption: $x^d_{e_1} \supseteq \cdots \supseteq x^d_{e_k}$. There is an even number of rows with 1 at position $x^d_{e_k}$ (since M'' must have columns with even sum). Each of these rows has 1 at each position $x^d_{e_j}, j < k$ as well, since each $x^d_{e_j}$ subsumes $x^d_{e_k}$, therefore contributing an even number to each $x^d_{e_j}$ column. Therefore, there must be an additional even number of rows with 1 at position $x^d_{e_{k-1}}$ (but 0 at position $x^d_{e_k}$), and so on, each $x^d_{e_j}$ contributing an even number of rows. Now consider a backward chain with representative back edge e_n. There must be even number of rows with 1's at position $x^d_{e_n}$, and these rows do not contain 1's at any other backward edge position $x^d_{e_{n'}}$. Each backward def-use chain contributes an even number of rows as well.

The proof of unimodularity of the constraint matrix of $LP_{\mathsf{Linear}(S)}$ follows by analogous arguments and is therefore omitted.

D.2 Proof for Theorem 3

Without loss of generality we consider the assignment of variables x and show correspondence between the x-assignment in $IP_{\mathsf{Linear}(S)}$ and the x-assignments in $IP_{C_{\text{MPC}}(S)}$.

Also without loss of generality, we focus on a single definition in $d \in C_{\text{MPC}}(S)$ and the constraints associated with d in both $IP_{\mathsf{Linear}(S)}$ and $IP_{C_{\text{MPC}}(S)}$. We assume that the problem presents the following constraints, grouped by (d, u) chains in categories (I) to (IV). Lemmas 1 and 2 in §5.1 entail that categories (I) to (IV) abstract away the structure of the system, and one can trivially generalize to an arbitrary number of (d, u) chains, i.e., categories.

Here $(d, u) \supseteq (d, u') \supseteq (d, u'')$ (also written as $e \supseteq e' \supseteq e''$, where e, e', and e'' are the corresponding representative edges), are forward def-use chains. (d, u^b) is a backward def-use chain, and we have that e' subsumes e^b, but e'' does not subsume e^b.

$$\begin{array}{lll}
 & IP_{\mathsf{Linear}(S)} & IP_{C_{\text{MPC}}(S)} \\
 & & \\
\text{(I)} & x^{\mathtt{st}_d} \geq a^{u} - a^{d} & x^d_e \geq a^u - a^d \\
\text{(II)} & x^{\mathtt{st}_d} \geq a^{u'} - a^{d} & x^d_e + x^d_{e'} \geq a^{u'} - a^d \\
\text{(III)} & x^{\mathtt{st}_d} \geq a^{u''} - a^{d} & x^d_e + x^d_{e'} + x^d_{e''} \geq a^{u''} - a^d \\
\text{(IV)} & x^{\mathtt{st}_d} \geq a^{u^b} - a^{d} & x^d_e + x^d_{e'} + x^d_{e^b} \geq a^{u^b} - a^d
\end{array}$$

Note that each category contains multiple constraints in the $IP_{\mathsf{Linear}(S)}$, where the $x^{\mathtt{st}_d}$'s are distinct. Each category contains a single constraint in the $IP_{C_{\text{MPC}}(S)}$, as shown.

Again, we prove the following useful lemmas before proof of the theorem. We give proofs sketches by considering a single illustrative case. The full proof is established by case-by-case analysis.

Lemma 3. *For each $(d, u) \in IP_{C_{MPC}(S)}$, there are exactly w_e distinct $x^{\mathtt{st}_d} \geq a^u - a^d$ constraints in $IP_{\mathsf{Linear}(S)}$, where $e = min\_cut(d, u)$.*

Proof. The above lemma states that for each def-use (d, u) there are exactly w_e constraints, where w_e is the weight of the min-cut edge of (d, u). Clearly, the number of constraints is given by $min(|\{\ \mathtt{st}^d \mid \alpha(\mathtt{st}^d) = d\ \}|, |\{\ \mathtt{st}^u \mid \alpha(\mathtt{st}^u) = u\ \}|)$. and the min-cut edge measures exactly that.

Lemma 4. $(d, u) \supseteq (d, u') \Rightarrow \gamma((d, u)) \supseteq \gamma((d, u'))$.

Proof. The second lemma states that when (d, u) subsumes (d, u') the set of $\mathtt{st}_d$'s associated with (d, u) includes all $\mathtt{st}_d$'s associated with (d, u'). As an informal argument, consider block B_1 immediately enclosed in block B_2, and let $d, u \in B_1$, and $u' \in B_2$ appear after B_1. Then only the $\mathtt{st}_d$ of the last iteration of B_1 is needed in constraints $x^{\mathtt{st}_d} \geq a^{\mathtt{st}_{u'}} - a^{\mathtt{st}_d}$; intuitively the definition in the last iteration "kills" all previous definitions, and is outwardly exposed to $u' \in B_2$.

Lemma 5. *Let (d, u) be a forward def-use chain, and (d, u^b) be a backward one. $(d, u) \not\supseteq (d, u^b) \Rightarrow \gamma((d, u)) \cap \gamma((d, u^b)) = \emptyset$.*

Proof. Again, consider block B_1 immediately enclosed in block B_2, and let $d, u^b \in B_1$, and $u \in B_2$. Since (d, u) does not subsume (d, u^b), u must appear in B_2, after B_1. The $\mathtt{st}_d$'s that are needed in backward def-use constraints $x^{\mathtt{st}_d} \geq a^{\mathtt{st}_{u^b}} - a^{\mathtt{st}_d}$; are all but the $\mathtt{st}_d$'s in the last iteration of B_1. (Since the last iteration cannot be used in $u^b \in B_1$.) In contrast, only the $\mathtt{st}_d$ of the last iteration of B_1 is needed in constraints $x^{\mathtt{st}_d} \geq a^{\mathtt{st}_u} - a^{\mathtt{st}_d}$ and we have that $\gamma((d, u))$ and $\gamma((d, u^b))$ are disjoint.

Proof. Let PA induce x such that for a fixed d, $\sum_e x_e^d \cdot w_e$ is minimal. We show that for the same fixed d, $\sum x^{\mathtt{st}_d} \geq \sum_e x_e^d \cdot w_e$ and then find values $x^{\mathtt{st}_d}$ that satisfy all constraints and $\sum x^{\mathtt{st}_d} = \sum_e x_e^d \cdot w_e$.

Recall categories (I)-(IV) above. We consider 3 cases.

Case (1) is when $x_{e'}^d$ is the "highest" def-use chain that requires conversion: $a^{u'} - a^d = 1$ (i.e., $a^u - a^d \geq 0$). Thus, $x_{e'}^d = 1$ and all $x_e^d, x_{e''}^d$, and $x_{e^b}^d$ are 0, or the sum will not be minimal. Therefore, $\sum_e x_e^d \cdot w_e = w_{e'}$ since all other terms in the sum are 0. Since $a^{u'} - a^d = 1$ we need all $x^{\mathtt{st}_d}$ in constraints $x^{\mathtt{st}_d} \geq a^{u'} - a^d$ (category (II)) to be set to 1. By Lemma 3, there are exactly $w_{e'}$ such constraints, and therefore, $\sum x^{\mathtt{st}_d} \geq w_{e'}$. By Lemma 4, $\gamma((d,u)) \supseteq \gamma((d,u')) \supseteq \gamma((d,u'')) \supseteq \gamma((d,u^b))$, and therefore category (III) and (IV) constraints are satisfied. We may set all $\gamma((d,u)) - \gamma((d,u')$ to 0, achieving $\sum x^{\mathtt{st}_d} \geq w_{e'}$.

Case (2) arises when $x_{e''}^d$, which does not subsume the backward chain, is the highest def-use chain that requires conversion, however, the backward chain $a^{u^b} - a^d \leq 0$. Then we have that all $x_e^d, x_{e'}^d$, and $x_{e^b}^d$ are 0, and by Lemma 3, $\sum x^{\mathtt{st}_d} = w_{e''}$.

Case (3) arises when $x_{e''}^d$ is the highest def-use chain that requires conversion, and the backward chain requires conversion as well, i.e., $a^{u^b} - a^d = 1$. Then one can easily see that the assignment that minimizes $\sum_e x_e^d \cdot w_e$ is x_e^d and $x_{e'}^d$ to 0, and $x_{e''}^d$ and $x_{e^b}^d$ to 1. Therefore, $\sum_e x_e^d \cdot w_e = w_{e''} + w_{e^b}$. There are exactly $w_{e''}$ constraints $x^{\mathtt{st}_d} \geq a^{u''} - a^d$ (category (III)) and w_{e^b} constraints $x^{\mathtt{st}_d} \geq a^{u^b} - a^d$ (category (IV)), and by Lemma 5, $\gamma((d,u'')) \cap \gamma((d,u^b)) = \emptyset$. Therefore, $\sum x^{\mathtt{st}_d} = w_{e''} + w_{e^b}$.

Although we consider only four categories, the system and proof can be trivially generalized to an arbitrary number of categories.

D.3 Optimal Assignment for $C_{\text{MPC}}(S)$ is also optimal for $\mathsf{Linear}(S)$

Assumptions

Uniform linearization of loops We assume that a block $\mathbf{B}$ in loop L—with upper bound N in $C_{\text{MPC}}(S)$—maps to a set of blocks $B_i, 1 \leq i \leq N$ in $\mathsf{Linear}(S)$. These B_i are all identical to $\mathbf{B}$ (modulo *pseudo-ϕ* nodes and variable names). Note that this assumption is natural, and not requiring it would mean proving optimality of $C_{\text{MPC}}(S)$ for arbitrary $\mathsf{Linear}(S)$ (instead of the $\mathsf{Linear}(S)$ that is linearization of $C_{\text{MPC}}(S)$).

Client-Server model restriction The parties performing the MPC are servers and inputs are given by the clients (a party may have both roles). We assume that inputs are received only *once* from the clients, by having them share there input(s) to the servers in a *single* type of sharing that the protocol specifies (*each input is shared once*). Then the servers compute the circuit on these shared values but cannot ask for more help from the clients. At the end the servers reconstruct the shared output and send to the clients. Note that this restriction only applies to input values from the clients, it does not include, for example, public constants.

Proof In the following we prove that, in non-amortized model, the constraints for $IP_{C_{\text{MPC}}(S)}$ are the same as those for $IP_{\mathsf{Linear}(S)}$. Therefore optimal assignment for $C_{\text{MPC}}(S)$ is also optimal for $\mathsf{Linear}(S)$. This is done on case by case analysis as under:

Case 1: Linear Code If $C_{\text{MPC}}(S)$ has no loops, then it is identical to $\mathsf{Linear}(S)$ which implies that constraints for $IP_{C_{\text{MPC}}(S)}$ are identical to constraints for $IP_{\mathsf{Linear}(S)}$. Therefore, solution of $IP_{C_{\text{MPC}}(S)}$ should be same as $IP_{\mathsf{Linear}(S)}$.

Case 2: Loop without pseudo-ϕ nodes Consider a $C_{\text{MPC}}(S)$ program that is a single loop L with N iterations. There are no pseudo-ϕ nodes in L. Let $\mathbf{B}$ denote the loop body block. In $\mathsf{Linear}(S)$, this program will result in a sequence of N blocks $B_1, B_2, \ldots, B_N$ where each $B_i, 1 \leq i \leq N$ is identical to $\mathbf{B}$ (modulo the variable names). Optimal assignment of each B_i will be exactly the same. This is because if some $B_j, j \neq i$ has better assignment than B_i, then that same assignment would benefit B_i as well. The total cost of $B_1, B_2, \ldots, B_N$ is therefore the cost of any B_i times N. Each statement s_{B_i} in B_i is essentially executed N times.

From case 1 above, we know that assignment for $\mathbf{B}$ and B_i will be the same (both are linear code). Furthermore, in $C_{\text{MPC}}(S)$, cost for each statement $\mathbf{s_B}$ in $\mathbf{B}$ is multiplied by $\mathbf{B}$'s weight (i.e. N here), Therefore $IP_{C_{\text{MPC}}(S)}$ will produce the same assignment and cost as $IP_{\mathsf{Linear}(S)}$ would.

Case 3: Loop with pseudo-ϕ nodes Now consider a $C_{\text{MPC}}(S)$ program with a block B_{pred}, followed by an N-iteration loop L with loop body $\mathbf{B}$, followed by a block B_{succ}. Without loss of generality, say L has a single pseudo-ϕ node i.e. there is a single definition $\mathbf{d}$ that $\mathbf{B}$ uses from previous iteration of the loop (i.e. $N-1$ iterations use $\mathbf{d}$). The first iteration of the loop will use a definition d' from B_{pred}.

This program will translate to $B_{pred}, B_1, B_2, \ldots, B_N, B_{succ}$ in $\mathsf{Linear}(S)$. The *pseudo-ϕ* node will disappear, and each $B_i, 1 < i \leq N$ will use definition d_i from B_{i-1}. B_1 will use d_1 from B_{pred}.

We already know from case 1 & 2 above that assignment and cost for $\mathbf{B}$ and B_i will be the same. However, now we may need to place conversions between iterations of $\mathbf{B}$ (i.e. between B_is).

In $\mathsf{Linear}(S)$, since all B_is will get the same assignment, the d_is that are produced in these blocks i.e. $d_i, 1 < i \leq N$ will be assigned the same sharing. Thus if any of these d_is needs conversion, all of them will need conversion. Therefore, we could simply multiply the cost of conversion for any d_i with $N-1$ to get the same resulting cost.

The definition d_1 used in B_1, however, is produced in B_{pred} (or any other preceding block for that matter), it may or may not need conversion. Therefore conversion constraint for d_1 will be separate from $d_i, 1 < i \leq N$.

In $C_{\text{MPC}}(S)$, conversion for $\mathbf{d}$ will have a weight of $N-1$, and for d', it will be 1 (since min-cut for d' to its use in L is the edge from B_{pred} to L's head). This is exactly the same as what we discussed for $\mathsf{Linear}(S)$ above.

Conclusion Using the above cases recursively, we can prove for arbitrary $C_{\text{MPC}}(S)$ programs that, in non-amortized cost model (and under the assumptions above), constraints for $IP_{C_{\text{MPC}}(S)}$ are the same as those for $IP_{\mathsf{Linear}(S)}$, Therefore optimal solution for $C_{\text{MPC}}(S)$ is also the optimal solution for $\mathsf{Linear}(S)$. $\square$

E Scheduling and Parallelization (Cont'd)

We next argue that a natural schedule meets the restrictions stated in §6.3, and therefore, the protocol assignment that minimizes $IP_{\mathsf{Linear}(S)}$ minimizes $IP_{\mathsf{Parallel}(S)}$ for a natural schedule.

First, we show that restriction (1) holds. Let $n \in C_{\text{MPC}}(S)$ be nested in D loops, k of which are parallel. Let $b_{i_1}, b_{i_2}, \ldots b_{i_k}$ be the bounds of the parallel loops, and let $b_{j_1} \ldots b_{j_{D-k}}$ be the bounds of the sequential loops. Then by construction, `st` nodes that map to n are grouped into $b_{j_1} \times \cdots \times b_{j_{N-k}}$ each group of size $b_{i_1} \times b_{i_2} \times \cdots \times b_{i_k}$. Therefore, the cost of n can be amortized over $b_{i_1} \cdot b_{i_2} \times \cdots \times b_{i_k}$ executions. For example, nodes that map to $n1$ in Fig. 3 are grouped in 2 groups each group of size 4.

Next, we sketch the argument that Restriction (2) also holds. The argument is by induction on the depth level of the def-use chain. Consider a def-use (d, u) and let B with bound b be the closest enclosing block of d and u; for simplicity, consider the case when both $d, u \in B$. If B is parallel, d's are grouped in 1 parallel node of size b; otherwise, d's are grouped in b nodes of size 1. Assume that after constructing the schedule at level k, d's are grouped in M parallel nodes, each of size S_M. If the $(k+1)$'st loop block, with bound b', is parallel, then d's remain grouped in M parallel nodes, each of size $b' \times S_M$ this time. Otherwise, i.e., if it is sequential, d's are grouped in $b' \times M$ nodes, each of size S_M. For example, consider the def-use $(n1, n6)$ in Fig. 3, whose closest enclosing block is B3. Before the linearization of B3, the definition $n1(B1_2)(B2_2)$ is in a single parallel node (it also contains other nodes that map to $n1$, however, those definitions are not exposed to $n6$). Since B3 is parallel, $n1(B1_2)(B2_2)(B3_1)$ and $n1(B1_2)(B2_2)(B3_2)$ are grouped in the same parallel node. The two definitions are shown in red in Fig. 3.

F Implementation and Benchmarks

This appendix goes into details of the implementation.

F.1 Analysis

As mentioned earlier, analysis phase takes a Java program as input and transforms it into MPC-source for analysis. The output of the analysis is a *def-use* graph that includes necessary information about nodes i.e. node types, their weight and their parallelizability.

Before we can build our def-use graph (and gather related information), we need to translate the input program into MPC-source. We use Soot to translate the input to Shimple, then we inline function calls (using wjop.si – an optimization pass built in to Soot –). Notice that inlining all calls can blow up the size of the entry routine (in our case main function) of the program. This makes heuristics or exhaustive-search based optimizations prohibitive because of the sheer number of choices in the analysis. This is not a problem in our case, as our optimization is the solution of a linear program. In fact, we benefit from inlining because it makes the analysis context sensitive.

Concretely, we perfrom the following analysis/transformations on the input program:

def-use chains We augment Soot's def-use analysis to handle arrays correctly. Soot's builtin def-use analysis works for scalars only. We cannot use it for arrays (vectors) because it treats array *writes* as a *use*. This is wrong in our context. For example, consider that the statement $v[i] = x + y$ is later followed by the statement $z = v[j] + w$. The first statement is an array write (it writes to v). The second statement is an array read (reads from v). In our context, the former is a *def* and the later is a *use*. Similar reasoning applies to multi-dimensional arrays (vectors), see figure 9 for an example. We treat each array *write* as a new definition.

```
1 r2 = newmultiarray (int)[100][4];
2 //... snip ...
3 $r8 = r2[i9_1];
4 $i6 = $r8[i11_2];
5 //... snip ...
6 $r8[i9_1] = i1;
7 //... snip ...
8 $r10 = r2[i13];
9 $i18 = $r10[i5_1];
```

Figure 9. Example Array Def-use Chains: the def-use chains for $r2$ are $(1, 4)$ and $(6, 9)$.

Mark Copies Given a statement like $x = y$, we tag x as a copy of y. This means marking all *uses* of x as uses of y and getting rid of x. This reduces the number of variables in the analysis, thereby simplifying it. As mentioned in §2, we can mark copies before collecting def-use chains since our IR is an SSA form. There is no particular reason to do it afterwards.

At this point, we have def-use chains (with no copies) and can start collecting additional information – node types, weights and conversion points – needed by the linear program. Node types, weights and conversion points are presented below.

Node Types As mentioned previously, the analysis maintains a mapping from Shimple operations to MPC gate types. In this step, it uses this mapping to assign a type to each node. For example node $x = y + z$ is an ADD type, or $x = y > z$ is a GT type.

Node Weights are computed exactly as described in §3.5.

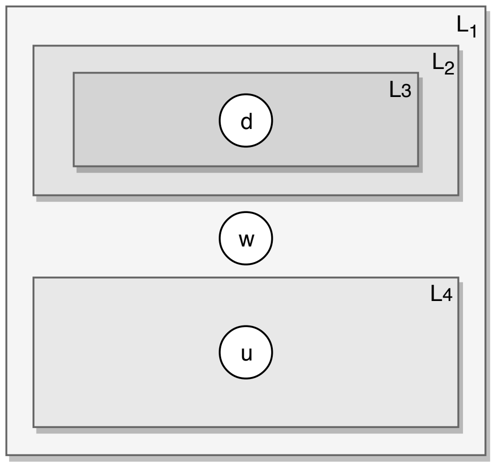


Figure 10. Conversion Point (min-cut): The conversion point (min-cut) for d (in L_3) and u (in L_4) is in their closest enclosing block L_1. w is one node where we can place conversion.

Conversion Points Conversions are needed if def-use (d, u) nodes are assigned different sharings. This entails computing *min-cute* on def-use chain as described in §3.6. In the implementation, this needs finding a node on the min-cute edge to use as location marker for conversion node.

We find optimal conversion point (min-cut) as follows. First, we construct a tree describing loop nesting. Then we find common ancestor of (d, u), say, L', which is the closest enclosing block as described in §3. Finally, for our conversion point, we find the closest edge e with target w to d in L'. Since this is a straight line program, we know that all paths from d to u pass through e and w. Fig. 10 illustrates this discussion visually.

At this point we have described def use chains, node types and weights, and conversion points, which is sufficient for protocol assignment in the sequential execution setting (we have established optimality in this setting). To make our analysis richer, we go one step further and compute parallelizability of nodes (i.e., a natural schedule as described in §6.2). This enables optimal protocol assignment in the parallel execution setting.

Node Parallelizability We use the following rule to determine if a loop L is parallelizable, essentially computing a schedule as described in §6.2. We compute def-use set S of all def-uses (d, u) that are *immediate*ly enclosed in L i.e. there exists no loop L' s.t. L' encloses S and L' is enclosed by L. Then we remove the def-use (d, u) chains corresponding to loop counter variables. Finally, for each def-use (d, u) chain in S, if transitive closure of any of d's *uses* contains d itself (i.e. d is used in the definition of itself in subsequent iteration), then L is not parallelizable. Fig. 11 illustrates an example with both a parallelizable and a non-parallelizable loop.

We exclude loop counter variables' def-use (d, u) chains from the above analysis. This is because such variables always depend on previous iterations of L and, therefore, transitive closure of such a d will always contain d. Thereby marking all loops (even the ones that are parallelizable), non-parallelizable.

If the above analysis yields that L is parallelizable, we mark all def-uses (d, u) in S as parallelizable assigning weights as described in §6.2.

Calculate Subsumption To compute subsumption (§5.1), we start at def d and create an empty ordered list. We now start processing d's successors with this list. If we find a use u, it is added to this list. Whenever control splits, we keep processing the fall-through successors as above. For branched successors we create a new list and recursively add any uses u' in the branch to this new list. At the end of it, we have collected one or more lists in which ordering indicates subsumption i.e. $\mathsf{index}(u) \leq \mathsf{index}(u') \implies (d, u) \supseteq (d, u')$.

```
for (int i=0; i<100; i++) {
    int sum = 0;
    for (int j=0; j<4; j++) {
        int diff = S[i][j] - C[j];
        int square = diff*diff;
        sum = sum + square;
    }
    D[i] = sum;
}
```

Figure 11. Checking Loop Parallelizablity: The outer loop is parallelizable but the inner is not (uses of sum include its definition).