COMBINE: COMpilation and Backend-INdependent vEctorization for Multi-Party Computation*

Benjamin Levy
levyb3@rpi.edu
Rensselaer Polytechnic Institute (RPI)
Troy, NY, USA

Muhammad Ishaq
ishaqm@purdue.edu
Purdue University
West Lafayette, IN, USA

Benjamin Sherman†
benjamin@bensherman.io
Rensselaer Polytechnic Institute (RPI)
Troy, NY, USA

Lindsey Kennard†
kennal@alum.rpi.edu
Rensselaer Polytechnic Institute (RPI)
Troy, NY, USA

Ana Milanova
milanova@cs.rpi.edu
Rensselaer Polytechnic Institute (RPI)
Troy, NY, USA

Vassilis Zikas
vzikas@purdue.edu
Purdue University
West Lafayette, IN, USA

ABSTRACT

Recent years have witnessed significant advances in programming technology for multi-party computation (MPC), bringing MPC closer to practice and wider applicability. Typical MPC programming frameworks focus on either *front-end* language design (e.g., Wysteria, Viaduct, SPDZ), or *back-end* protocol design and implementation (e.g., ABY, MOTION, MP-SPDZ).

We propose a methodology for an MPC compilation toolchain, which by mimicking the compilation methodology of classical compilers enables *middle-end* (i.e., *machine-independent*) optimizations, yielding significant improvements. We advance an intermediate language, which we call *MPC-IR* that can be viewed as the analogue of (enriched) Static Single Assignment (SSA) form. MPC-IR enables backend-independent optimizations in a close analogy to machine-independent optimizations in classical compilers. To demonstrate our approach, we focus on a specific backend-independent optimization, SIMD-vectorization: We devise a novel classical-compiler-inspired automatic SIMD-vectorization on MPC-IR. To demonstrate backend independence and quality of our optimization, we evaluate our approach with two mainstream backend frameworks that support multiple types of MPC protocols, namely MOTION and MP-SPDZ, and show significant improvements across the board.

CCS CONCEPTS

• **Security and privacy → Cryptography**; **Software and application security**.

KEYWORDS

multi-party computation, compiler optimizations

*Full version of this work is available on eprint [39].
†Work done while the author was a graduate student at RPI.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '23, November 26–30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623181

ACM Reference Format:
Benjamin Levy, Muhammad Ishaq, Benjamin Sherman, Lindsey Kennard, Ana Milanova, and Vassilis Zikas. 2023. COMBINE: COMpilation and Backend-INdependent vEctorization for Multi-Party Computation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623181

1 INTRODUCTION

Multi-party computation (MPC) allows N parties $P_1, \dots, P_N$ to perform a computation on their private inputs securely. Informally, security means that the secure computation protocol computes the correct output (correctness) and it does not leak any information about the individual party inputs beyond what can be deduced from the output (privacy).

MPC theory dates back to the early 1980s [54, 31, 9, 19]. Long in the realm of theoretical cryptography, MPC has seen significant advances in application in recent years. New tools and compilers bring MPC closer to practice and wider applicability, e.g., [13, 11, 42, 41]. The goal is to enable programmers to write *secure* and *efficient* programs without commanding extensive knowledge of cryptographic primitives.

Recent advances in MPC programming technology tend to focus on either frontend language design (e.g., Wysteria [48], Wys* [50], and Viaduct [1]) or backend circuit/protocol design and implementation (e.g., SPDZ family [37, 6, 36], MOTION [14]). The former, frontend-focused thread devised high-level constructs to express multiple parties, computation by different parties, and information flow from one party to another [48, 50, 1]. The latter, backend-focused thread devised cryptographic protocols, typically at the circuit-level [25, 6, 14, 47, 37].

In this work we focus on the *middle end*. We formalize an intermediate representation (IR) tailored to MPC, called *MPC-IR*, and focus on what we call *backend-independent* optimizations, a close analogue to *machine-independent* optimization in the classical compiler. Fig. 1 depicts our position in the compiler stack. We envision different front ends compiling into MPC-IR — the frontend we use in this work is *IMP Source*, an easy to use Python-like language.

MPC-IR exposes the *linear structure* of MPC programs, which simplifies program analysis; this is in contrast to IMP source, which has branching constructs. At the same time, MPC-IR is sufficiently "high-level" to support analysis and optimizations that take into

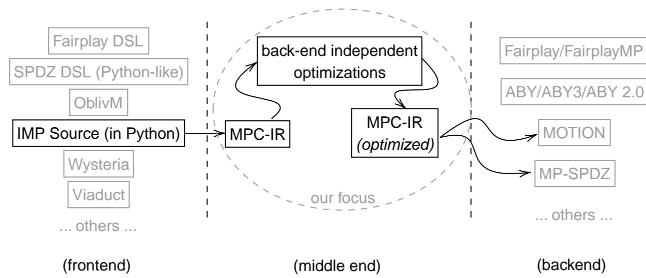

Figure 1: Our focus is middle end of the compiler stack.

account control and data flow in a specific program. As an added benefit, MPC-IR facilitates simple and abstract modeling of (amortized) cost associated to different operations; this makes it suitable for defining optimizations that take advantage of amortization at the circuit level such as SIMD-vectorization and protocol mixing [17, 34, 29, 28, 20]. Our formalization enables reasoning about optimizations over MPC-IR and correctness proofs of the optimizations and transformations. We demonstrate the utility of the IR and optimizations on two MPC backends, MOTION [14] and MP-SPDZ [36]. We include a more detailed exposition and comparison with related work in §3.

Our Contribution. We describe an end-to-end compiler framework that takes a Python-like routine (frontend) and produces optimized backend code. More concretely we present (a) our (Python-like) IMP Source language, its syntax and semantic restrictions; (b) formal specification of MPC-IR; (c) a translation of IMP Source into MPC-IR, (d) a specific backend-independent optimization: novel SIMD-vectorization on MPC-IR; and (e) translation of MPC-IR into MOTION and MP-SPDZ code for a large set of benchmarks.

We also provide an analytical model for cost estimation of amortized schedules. Our model simplifies the problem of cost estimations by abstracting away several of the complexities. We note in passing that such cost modeling is important as it drives not only vectorization but also optimizations such as protocol mixing and scheduling [34, 29, 20]. Given the restrictions on MPC programs that are imposed by privacy requirements—e.g., the linear program structure and bounded loops—one might think that this problem is simpler for MPC than the classical scheduling problem (which is known to be NP-hard). One might ask: *Is cost estimation for amortized schedules in MPC (in this model) tractable?* We answer this question to the negative: We show that even in our cost model, the problem remains NP-Hard; we show this via a reduction to the Shortest Common Supersequence (SCS) problem.

The above demonstrates that optimized MPC scheduling is an interesting problem. In this work, we provide an optimization by utilizing our cost model and IR and taking inspiration from the classical area of high-performance computing (HPC): a common technique there is *vectorization*, aka "SIMDification". Informally, a Single Instruction, Multiple Data (in short, SIMD) operation works with vectors of data instead of scalars, and replaces N operations on scalars with a single operation on vectors of size N.

In more detail, we provide a novel technique for automatic vectorization of MPC-IR programs. Vectorization not only reduces running time, but also reduces communication by enabling better packing. If the backend MPC framework supports SIMD gates—most mainstream backends do, in particular the ones chosen here—this results in smaller circuits and reduces both the time and memory footprint of circuit generation. We note in passing that although much of the applied MPC literature has focused on run-time improvements, for the technology to become mainstream it is imperative to also improve the compilation time. Our work demonstrates improvements in both.

We demonstrate the wide applicabiliy of our framework (and the expressivity of the source language) by running the compiler on 15 programs with interleaved if- and for-statements. We further demonstrate the utility and the backend-independent nature of our optimization by generating iterative and vectorized code for two well-known backend frameworks, MOTION [14] and MP-SPDZ [36], taking advantage of the respective SIMD API in vectorized code; each backend then translates the code into circuits and runs the circuits. We chose these two frameworks as they are not only among the more prominent in the relevant literature, but they are also highly parameterizeable as they support several MPC paradigms. They can generate and evaluate Boolean or Arithmetic circuits with GMW-style protocols [31] or Boolean circuits with BMR-style protocols [7]—for brevity we refer to the corresponding protocols as (Boolean/Binary or Arithmetic) GMW and BMR. This not only allows us to showcase our optimization in a wide range of benchmarks—across different MPC paradigms (see below)—but also opens our methodology to what we view as a major application area for our optimizer, namely MPC mixing.

In more detail, for MOTION we generate code for Boolean GMW and BMR—we do not include Arithmetic GMW, as MOTION does not support all operations in the Arithmetic GMW protocol, e.g., MUX, and we do not yet support protocol mixing. Towards evaluation we run experiments with two parties (2PC) and three parties (3PC).[1] *Circuit evaluation time* for vectorized code improves over iterative code up to 30x for 2PC (resp. 40x for 3PC setting) in GMW and up to 45x for 2PC (resp. 55x for 3PC) in BMR. For the operations that do not depend on number of parties, *communication size* reduces by up to 13x in GMW and 3x in BMR. Similarly, *circuit generation time* and *number of gates* reduce, respectively, by up to 200x and 480x in GMW, and 80x and 450x in BMR.

For MP-SPDZ we generate Python code then compile and execute in the Arithmetic setting and in the Binary setting. We run experiments with 2PC and demonstrate reduction in compilation time of up to 25x in the Binary setting.

Full source code and benchmarks are available at https://github.com/milana2/ParallelizationForMPC where https://github.com/milana2/ParallelizationForMPC/tree/gh-pages shows experiments on small input sizes and details the different stages of the compiler.

Our results emphasize the opportunities of backend-independent optimizations; we believe that our work can lead to future work on backend-independent compilation and new optimizations for MPC.

[1]The framework has no restriction on number of parties or MPC paradigm (active/passive) as long as the underlying backend has necessary support.

Outline. The rest of the paper is organized as follows. §2 presents an overview of our techniques. §3 reviews the related work. §4 describes the cost estimation model and argues intractability of optimal scheduling. §5 details the frontend of the compiler, §6 focuses on MPC-IR and backend-independent vectorization, and §7 briefly describes translation into MOTION and MP-SPDZ. §8 presents the experiments and §9 concludes the paper.

2 OVERVIEW OF METHODOLOGY

Below, we provide an overview of our methodology using the standard MPC benchmark of Biometric matching as an example.

IMP-Source as MPC Source Code. Listing 1(a) shows an intuitive implementation of Biometric matching. Array C is the feature vector that we wish to match and S is the database of N size-D vectors that we match against.

Our compiler takes essentially standard IMP [45] syntax and imposes certain semantic restrictions (details will follow). The programmer writes an iterative program and annotates certain inputs and outputs as *shared*. In the example, arrays C and S are `shared`, meaning that they store shares (secrets), however, the array sizes D and N respectively are plaintext. The code iterates over S and computes the Euclidean distance of the current entry S[i] and C (its square actually).

MPC-IR and Schedule Cost. Our compiler generates MPC-IR, a *linear* Static Single Assignment (SSA) form. Listing 1(b) shows the translation of the code in 1(a) into MPC-IR. We provide background on SSA and automatic vectorization from the point of view of compilation and optimization for MPC in an Appendix in the full version of our paper [39]; we refer interested readers to the full version if they are unfamiliar with these notions from compiler theory.

We turn to our analytical model to compute the *cost* of the iterative program. Assume cost β for a local MPC operation (e.g., XOR in Boolean sharing or ADD in Arithmetic sharing) and cost α for a remote MPC operation (e.g., MUX, CMP, etc.). Assuming that ADD is β and SUB, CMP and MUX are α, the MPC-IR in Listing 1(b) gives rise to an iterative schedule with cost $ND(2\alpha + \beta) + N(3\alpha)$.

Vectorized MPC-IR and Schedule Cost. We can compute all N times D subtraction operations at line 9 in 1(b) in a single SIMD instruction; similarly we can compute all multiplication operations at line 10 in a single SIMD instruction. Our compiler runs Listing 1(b) through the vectorization optimization to produce 1(c). Note that this is still our IR, Optimized MPC-IR. The compiler turns this code into variables, loops and SIMD primitives (if supported), suitable for the backend to generate the circuit.

In MPC backends, executing n operations "at once" in a single SIMD operation costs less than executing those n operations one by one. This is particularly important for interactive gates, since it allows many 1-bit values to be sent at once. We consider that each operation has a *fixed* portion that benefits from amortization and a *variable* portion that does not benefit from amortization: $\alpha = \alpha_{fix} + \alpha_{var}$. This gives rise to the following formula for amortized cost: $f(n) = \alpha_{fix} + n\alpha_{var}$, as the fixed costs of n operations are combined into one. This is in contrast to the the unamortized cost: $g(n) = n\alpha_{fix} + n\alpha_{var}$. (We extend the same reasoning to β-instructions.)

Thus, the fixed cost of the vectorized program amounts to $2\alpha_{fix} + D\beta_{fix} + N(3\alpha_{fix})$. The variable cost is the same in both the vectorized and non-vectorized programs. The first term in the sum corresponds to the vectorized subtraction and multiplication (lines 9-10 in (c)), the second term to the for-loop on j (lines 12-16), and the third one to the remaining for-loops on i (lines 19-25). Clearly, $2\alpha_{fix} + D\beta_{fix} + N(3\alpha_{fix}) << ND(2\alpha_{fix} + \beta_{fix}) + N3\alpha_{fix}$. Empirically, we observe significant improvement, e.g., in the MOTION experiments Biometric matching evaluation time improves 10x (GMW) and 23x (BMR) in 2PC, and 12x (GMW) and 28x (BMR) in 3PC.

3 RELATED WORK

MPC Compilers. The early MPC compilers—Fairplay [8] and Sharemind [12]— demonstrated that MPC can be brought to applications. This led to the development of a diverse landscape of implementations/compilers for specific MPC protocols, e.g., PICCO [56], Obliv-C [55], TinyGarble [52], Wystiria [48], Frigate [43], SPDZ/SPDZ-2 [24, 23, 38], SCALE-MAMBA [21] and others. A new generation of state-of-the art (and actively developed) MPC backends includes MP-SPDZ [36] and the ABY/HyCC/MOTION [25, 17, 14] frameworks. These frameworks are highly parameterizable and allow the use of different MPC protocols/paradigms. Another recent development is Viaduct, a language and compiler that supports a range of secure computation frameworks, including MPC and ZKP. For a (by now slightly outdated) review of MPC compiler frameworks please see [32]. In contrast to the above works, we focus on backend-independent optimizations available at the "higher-level" MPC-IR. Our key goal is to demonstrate provably correct compiler optimization for MPC, not a language frontend. Indeed, replacing our frontend with a more functional frontend, such as Viaduct's, is an interesting future direction.

Obliv-C [55], Wysteria [49] and Viaduct [1] focus on higher-level language design. ObliVM [40] has similar goals to ours but our works are complementary in the sense that while ObliVM relies on programmer annotations such as map-reduce constructs, we automatically detect opportunities for optimization at an intermediate level of representation. Similarly, GraphCG [44] introduces high-level programming abstractions and novel parallel oblivious algorithms; programmers can make use of the abstractions to write highly parallel and efficient secure algorithms.

HyCC [17] is a compiler from C Source into ABY circuits. It does source-to-source compilation with the goal to decompose the program into modules and then assign protocols to modules. In contrast, we focus on MPC-IR-level optimizations, specifically vectorization, although we envision future optimizations as well. HyCC, similarly to Buscher et al. [18] and [16], uses an off-the-shelf source-to-source polyhedral compiler [2] to perform vectorization at the level of source code. The disadvantage of using an off-the-shelf source-to-source compiler is that it solves a more general problem than what MPC presents and may forgo optimization because polyhedral compilation does not work well with conditionals (see [10]). In addition, the MPC-IR produced by our vectorization algorithm

[2] To our understanding, HyCC uses Par4All (https://github.com/Par4All/par4all), however, Par4All does not appear to be included with HyCC's publicly available distribution.

```
1  def biometric(C: shared[list[int]], D: int,
2          S: shared[list[int]], N: int) ->
3          shared[tuple[int,int]]:
4      min_sum : int = MAX_INT
5      min_idx : int = 0
6      for i in range(N):
7          sum : int = 0
8          for j in range(D):
9              # d = S[i,j] - C[j]
10             d : int = S[i * D + j] - C[j]
11             p : int = d * d
12             sum = sum + p
13         if sum < min_sum:
14             min_sum : int = sum
15             min_idx : int = i
16     return (min_sum, min_idx)
```

(a) IMP Source

```
1  min_sum!1 = MAX_INT
2  min_idx!1 = 0
3  for i in range(0, N):
4      min_sum!2 = PHI(min_sum!1, min_sum!4)
5      min_idx!2 = PHI(min_idx!1, min_idx!4)
6      sum!2 = 0
7      for j in range(0, D):
8          sum!3 = PHI(sum!2, sum!4)
9          d = SUB(S[((i * D) + j)],C[j])
10         p = MUL(d,d)
11         sum!4 = ADD(sum!3,p)
12     t = CMP(sum!3,min_sum!2)
13     min_sum!3 = sum!3
14     min_idx!3 = i
15     min_sum!4 = MUX(t, min_sum!3, min_sum!2)
16     min_idx!4 = MUX(t, min_idx!3, min_idx!2)
17 return (min_sum!2, min_idx!2)
```

(b) MPC-IR

```
1  min_sum!1 = MAX_INT
2  min_idx!1 = 0
3  # S^ is same as S. C^ replicates C N times:
4  S^ = raise_dim(S, ((i * D) + j), (i:N,j:D)) #S^[i,j] = S[i,j]
5  C^ = raise_dim(C, j, (i:N,j:D)) #C^[i,j] = C[j]
6
7  sum!2[I] = [0,..,0]
8  # computes _all_ "at once"
9  d[I,J] = SUB_SIMD(S^[I,J],C^[I,J])
10 p[I,J] = MUL_SIMD(d[I,J],d[I,J])
11
12 for j in range(0, D):
13     # sum!2[I], sum!3[I], sum!4[I] are size-N vectors
14     # computes N intermediate sums "at once"
15     sum!3[I] = PHI(sum!2[I], sum!4[I])
16     sum!4[I] = ADD_SIMD(sum!3[I],p[I,j])
17
18 min_idx!3[I] = [0,1,...N-1]
19 for i in range(0, N):
20     min_sum!2 = PHI(min_sum!1, min_sum!4)
21     t[i] = CMP(sum!3[i],min_sum!2)
22     min_sum!4 = MUX(t[i], sum!3[i], min_sum!2)
23 for i in range(0, N):
24     min_idx!2 = PHI(min_idx!1, min_idx!4)
25     min_idx!4 = MUX(t[i], min_idx!3[i], min_idx!2)
26 return (min_sum!2, min_idx!2)
```

(c) Optimized MPC-IR

Table 1: Biometric Matching from IMP Source to Optimized MPC-IR. MPC-IR is an SSA form without conditionals, therefore the conditional on lines 13-15 in (a) turns into linear code on lines 12-16 (b). In (c), our compiler fully vectorizes the SUB and MUL operations on lines 9 and 10 of (b). The computation of sum (line 11 in (b)) is sequential across the j-dimension, but it is parallel across the i-dimension as the loop on lines 12-16 in (c) illustrates.

can serve as input to protocol mixing algorithms, such as, for example, OPA [34], which requires vectorized input and appears to use ad-hoc and manual vectorization.

Intermediate representations for MPC. Recent work makes progress on intermediate representations for MPC. Ozdemir et al. [46] develop CirC, an IR with backends into ZKP and SMT primitives. MPC-IR is a higher level of representation than CirC; specifically, it does not unroll loops, leading to more scalable analysis. Heldmann et al. [33] present an LLVM-IR-based toolchain for compilation into circuits. We take a different approach — rather than reverse engineer LLVM-IR, which is rich and complex, we propagate necessary information from the frontend to the middle-end IR; in addition, we formalize MPC-IR to enable reasoning about correctness of transformations. Another recent IR, developed concurrently and independently from ours, is FUSE [15]. It enables optimizations such as vectorization, in a way similar to the Section 6.1 optimization in MP-SPDZ [36] (to our understanding). MPC-IR is a higher level IR and is orthogonal to FUSE. One can compile MPC-IR into FUSE and take advantage of the optimizations available at this level; our results with MP-SPDZ demonstrate that the two IRs and optimizations complement each other. In addition, we present syntax and semantics of MPC-IR, which enables reasoning about code transformations.

Automatic vectorization in HPC. Automatic vectorization is a longstanding problem in high-performance computing (HPC). We present a vectorization algorithm for MPC-IR that builds upon classical loop vectorization by Allen and Kennedy [4]. We view Karrenberg's work on Whole function vectorization [35] as most closely related to our work — it linearizes the program and vectorizes both branches of a conditional applying masking to avoid execution of the branch-not-taken code, and selection (similar to MUX). We argue that vectorization over linear MPC-IR is a problem that warrants a new look, while drawing from results in HPC. Since both branches of the conditional and the multiplexer *always* execute, not only can one apply aggressive vectorization on linear code, but (perhaps more importantly) one can build analytical models that accurately predict execution time. The models can drive optimizations such as vectorization, protocol mixing and others; these optimizations interact in non-trivial ways.

Polyhedral parallelization [10] is another rich area. It considers a higher-level source (typically AST) representation, while, in contrast, our work takes advantage of linear MPC-IR and SSA, and the corresponding dependence analysis. Karrenber's work [35] is rare in that space, in the sense that it considers vectorization over SSA, which has similarities with analysis over MPC-IR. We differ in the array representation, notion of dependence, and reasoning about dependence, as we specifically target MPC-IR.

4 ANALYTICAL (PARALLEL) COST MODEL

Next, we detail our model for cost estimation of MPC schedules, and show that optimal scheduling (for MPC) is NP-Hard.

4.1 Scheduling in MPC

We work in a single protocol setting i.e., all MPC tasks are evaluated in a single protocol from start to finish. In addition, we abstract common features of MPC execution in the following assumptions:

First, there are two types of MPC instructions, local and remote. A local instruction (i.e., ADD or XOR) has cost β and a remote instruction (i.e. MUL) has cost α, where $\alpha >> \beta$. We assume that all remote instructions have the same cost α and all local ones have the same cost β. We remark more on this later.

Second, in MPC frameworks, executing n operations "at once" in a single SIMD operation costs less than executing those n operations one by one. Following Amdahl's law [5], we write $\alpha(s) = \frac{1}{s}p\alpha + (1-p)\alpha$, where p is the fraction of execution time that benefits from amortization and $(1-p)$ is the fraction that does not, and s is the available resource. Thus, $n\alpha(s) = \frac{n}{s}p\alpha + n(1-p)\alpha$. For the purpose of the model we assume that s is large enough and the term $\frac{n}{s}p\alpha$ amounts to a *fixed cost* incurred regardless of whether n is $10,000$ or just 1. (This models the cost of preparing and sending a packet from party A to party B for example.) Therefore, amortized execution of n operations is $f(n) = \alpha_{fix} + n\alpha_{var}$ in contrast to unamortized execution $g(n) = n\alpha_{fix} + n\alpha_{var}$. We have $\alpha_{fix} << n\alpha_{fix}$ and since fixed cost dominates variable cost (particularly for remote operations), we have $f(n) << g(n)$.

Third, MPC instructions scheduled in parallel benefit from amortization *only if* they are the same instruction. Given our previous assumption, 2 MUL instructions can be amortized in a single SIMD instruction that costs $\alpha_{fix} + 2\alpha_{var}$, however a MUL and a MUX still cost $2\alpha_{fix} + 2\alpha_{var}$ even when scheduled "in parallel".

4.2 (Intractability of) Optimal MPC Scheduling

Given a schedule of an MPC program i.e. a sequence of instructions $S := (S_1; \dots; S_n)$, and a def-use dependency graph $G(V,E)$ corresponding to S, our task is to construct a parallel schedule $P := (P_1; \dots; P_m)$ such that:

- All P_i's consist of instructions of the same kind.
- Def-use dependencies of the graph $G(V,E)$ are preserved i.e. if instructions $S_i, S_j, i < j$ form a def-use i.e. an edge exists from S_i to S_j in G, then they can only be mapped to $P_{i'}, P_{j'}$ such that $i' < j'$.

Correctness of P follows due to the preservation of def-use *dependencies.* One can easily argue by induction on the length of schedule S that the computed function is the same in both S and P.

The cost of schedule S is

$$cost(S) = \sum_{i=1}^{n} cost(S_i) = L_\alpha \alpha_{fix} + L_\beta \beta_{fix} + L_\alpha \alpha_{var} + L_\beta \beta_{var} \quad (1)$$

where L_α is the number of α-instructions and L_β is the number of β ones. (We used this formula to compute the cost of the unrolled MPC-IR program in §2.) The cost of schedule P is more interesting:

$$cost(P) = \sum_{i=1}^{m} cost(P_i) \quad (2)$$

Each P_i may contain multiple instructions, and $cost(P_i)$ is amortized. Thus, according to our model $cost(P_i) = \alpha_{fix} + |P_i|\alpha_{var}$ if P_i stores $|P_i|$ α-instructions, or $cost(P_i) = \beta_{fix} + |P_i|\beta_{var}$ if it stores β-instructions. (Similarly, we used this formula to compute the cost of the Optimized MPC Source program in §2.)

Our goal is to construct a parallel schedule P that reduces the program cost (when compared to cost of S). One would hope that simpler MPC program structure would make optimal scheduling tractable. Intuitively, the problem is to combine multiple independent schedules (or sequences of instructions) into a single schedule where same instructions are scheduled into a SIMD-instruction P_i. This amounts to finding a Shortest Common Supersequence for the independent schedules. We formalize the argument below and show that the scheduling problem is NP-hard via a reduction to the Shortest Common Supersequence problem [53].

Proof. To argue that optimal scheduling is an NP-Hard problem, we consider the following convenient representation. An MPC program is represented as a set of sequences $\{s_1, \dots, s_n\}$ of operations. In each sequence s_i operations depend on previous operations via def-use relations, i.e. $s_i[j], j > 1$ depends on $s_i[j-1]$.

As an example, consider the MPC program consisting of the following three sequences, all made up of two distinct α-instructions M_1 and M_2, e.g., M_1 is MUL and M_2 is MUX. The right arrow indicates a def-use *dependence*, meaning that the source node must execute before the target node:

(1) $M_1 \rightarrow M_2 \rightarrow M_1$
(2) $M_1 \rightarrow M_1 \rightarrow M_1$
(3) $M_2 \rightarrow M_1 \rightarrow M_2$

The problem is to find a schedule P with *minimal cost.* For example, one such schedule for the sequences above is

$M_1(1)||M_1(2)$; $M_1(2)$; $M_2(1)||M_2(3)$; $M_1(1)||M_1(2)||M_1(3)$; $M_2(3)$

The parentheses above indicate the sequence where the instruction comes from: (1), (2), or (3). Cost of schedule P is computed using Eq. (2) and it amounts to $5\alpha_{fix} + 9\alpha_{var}$.

The problem of finding a schedule P with a minimal $cost(P)$ is shown to be NP-Hard problem, as it can be reduced to the problem of finding a *shortest common supersequence*, a known NP-Hard problem [53]. The shortest common supersequence problem is as follows: *given two or more sequences find the the shortest sequence that contains all of the original sequences.* This can be solved in $O(n^k)$ time, where n is the cardinality of the longest sequence and k is the number of sequences. We can see that the optimal schedule is the shortest schedule, since the shortest schedule minimizes the fixed cost while the variable cost remains the same.

To formalize the reduction, suppose P is a schedule with minimal cost (computed by a black-box algorithm). Clearly P is a supersequence of each sequence s_i, i.e., P is a common supersequence of $s_1 \dots s_n$. It is also a shortest common supersequence. The cost of $cost(P) = L\alpha_{fix} + N\alpha_{var}$ where L is the length of P and N is the total number of instructions across all sequences. Now suppose, there exist a shorter common supersequence P' of length L'. $cost(P') < cost(P)$ since $L'\alpha_{var} + N\alpha_{var} < L\alpha_{var} + N\alpha_{var}$, contradicting the assumption that P has the lowest cost. □

Finally, we remark on the (necessity of) assumptions. The assumption that "there are only two types of instructions, α and β" comes in the proof only, and it comes w.l.o.g. as to prove NP-hardness for an arbitrary circuit (that uses higher-level gates), one can first reduce this new circuit to another one that only uses MUL and ADD gates and then use our proof. Since the reduction is polynomial time, this proof will imply NP-hardness of optimal scheduling for the new (lower-level) circuit. We stress that this assumption is for the NP-hardness proof only. The vectorization optimization does

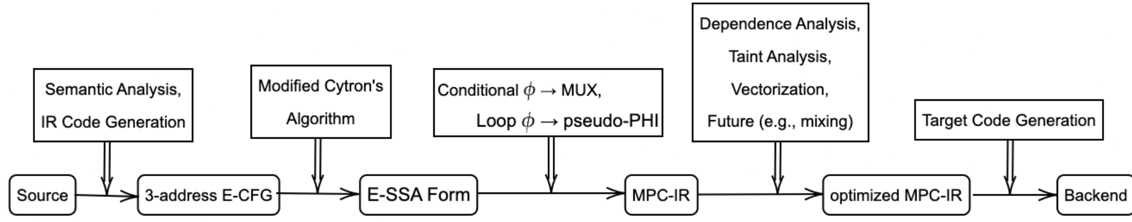

Figure 2: Compiler Framework.

not take into account costs — it vectorizes every operation it can — so there is no negative impact there. Notwithstanding, since this assumption is w.l.o.g., for optimizations that actually use the cost tables, e.g., mixing, one can use a refined model and it will not affect the proof.

The assumption that "only same-type instructions benefit from vectorization" is a simplifying assumption; it is not strictly true, but assuming it, as in e.g., [34, 25, 41], helps simplify the problem. One might be able to get additional improvements by identifying parallelizable heterogeneous instructions, but it is unclear how to do this in an automated compiler without unrolling loops (unrolling loops would go a level of abstraction below MPC-IR and would slow down the compilation process and increase the memory needed by the compiler to store the unrolled loop information).

5 COMPILER FRONTEND

This section presents an overview of our compiler, followed by our source syntax and semantic restrictions.

5.1 Overview

Fig. 2 shows the phases of our compiler. We present a systematic translation of high-level source, which includes if-then-else statements and for loops, into a linear sequence of primitive MPC instructions. We start with an Abstract Syntax Tree (AST) source syntax, then convert it into a Control-flow graph (CFG), as Cytron's classical SSA algorithm [22] is defined over a CFG. SSA is the natural means for converting if-then-else statements into MUX primitives, as there is correspondence between ϕ-nodes and MUX primitives. Our conversion takes advantage of the properties of Cytron's SSA, particularly the *minimal number*[3] of ϕ-nodes that results in a minimal number of MUX primitives. We then translate into MPC-IR, which is most conveniently described with an AST syntax (cf. §6.4).

We also note that we settled on this process after we failed at the straightforward approach. Initially, we attempted to reuse existing implementations of SSA intermediate representations such as Soot Shimple (http://soot-oss.github.io/soot/) or LLVM IR (https://llvm.org/). The problem was that these were CFG representations and they had lost connection between the ϕ-node and the conditional that triggered the ϕ-node. Specifically, given $x = \phi(y, z)$, there is no information what conditional triggered the ϕ-node and whether y corresponds to the true or false branch of the conditional. Moreover, ϕ-nodes with 3 or more arguments are common in standard SSA. Furthermore, existing SSA IRs do not handle arrays, while handling of arrays is important for vectorization. As a remark, while it is possible to reconstruct the missing information via control dependence analysis (e.g., as in [33]), and it is possible to add arrays, this proves very difficult due to the complexity of the IRs (these IRs are designed to handle richer and more complex syntax than MPC). A clean state solution, where we start from the AST and retain all necessary information in the CFG i.e. enhanced-CFG (or E-CFG in Fig. 2) and construct enhanced-SSA (or E-SSA) that handles arrays, proves the correct choice for our compiler and drives our progress. Specifically, E-CFG and E-SSA extend ordinary CFG and SSA by adding a *merge condition* property to conditional ϕ-nodes. This property contains the condition variable of the if-statement that created the branch that the ϕ-node is merging. This naturally gives rise to MPC-IR, where conditional ϕ-nodes translate to MUX nodes, and loop ϕ-nodes translate into what we call pseudo PHI-nodes (lines 4, 5 and 8 in Listing 1(b)). Details on the translation from source to E-SSA, and from E-SSA to MPC-IR are in the full version of the paper [39].

[3]A naive SSA translation may place a ϕ-node for every variable at every control merge node, but Cytron's SSA places ϕ-nodes only where they are needed, at the Dominance Frontier of variable assignments.

5.2 Syntax and Semantic Restrictions

Source syntax is standard IMP syntax but with for-loops:

$e ::= e\ op\ e \mid$ x $\mid$ const $\mid$ A[e]	*expression*
$s ::= s; s \mid$	*sequence*
x = e $\mid$ A[e] = e $\mid$	*assignment stmt*
for i in range(I) : s $\mid$	*for stmt*
if e: s else: s	*if stmt*

The syntax allows for array accesses, arbitrarily nested loops, and if-then-else control flow.

The prototype version of the compiler assumes the following semantic restrictions on source programs. Currently, the compiler does not enforce the restrictions as the majority of them are implementation restrictions that can be easily lifted. They can be easily encoded as rules in a syntax-directed translation [2, 51] over the syntax above.

- Loops are of the form $0 <= i < I$ and bounds are fixed at compile time. It is a standard restriction in MPC that the bounds must be known at circuit-generation time.
- Arrays are one-dimensional. N-dimensional arrays are linearized and accessed in row-major order.
- Array subscripts are plaintext. We will lift this restriction in future work by applying the standard linear scan [40, 6] when the subscript is secret-shared.

- The subscript e is a function of the indices of the enclosing loops. For read access, the compiler allows an arbitrary such function. However, it restricts write access to *canonical access*, i.e., $\mathsf{A}[i, j, k] = \ldots$ where i, j and k are the indices of the *outermost* loops enclosing the array write statement. These indices loop over the three dimensions of A and all write accesses to A follow this restriction. We note that this is a restriction on the vectorization optimization; there is no reason to restrict arbitrary writes in the code, they just are not optimized. We state this restriction upfront as it simplifies vectorization and its exposition.
- The final restriction disallows array writes from within if-then-else statements. This is to ensure that arguments of MUX in the MPC-IR translation are base types, i.e., just int or bool. In our experience, this causes a minor inconvenience to the programmer as they may not write

```
1 if e: A[i] = val
```

Instead they write (simplifying slightly)

```
1 if not(e): val = A[i]
2 A[i] = val
```

In addition, our compiler defines and implements a taint type system at the level of MPC-IR. The system, which is a standard positive-negative type qualifier system, is detailed in the full paper. We note that while the programmer writes annotations at the level of IMP Source (as in Listing 1(a)), the annotations propagate through the transformations. The only required annotations are on the input arguments; the rest are inferred and checked with a standard taint analysis (based on the type system) at the level of MPC-IR.

For the rest of this paper we write i, j, k to denote the loop nesting: i is the outermost loop, j, is immediately nested in i, and so on until k and we use I, J, K to denote the corresponding upper bounds. We write $\mathsf{A}[i, j, k]$ to denote canonical access to an element, either array element or a scalar expanded to its loop nest i, j, k. To simplify the presentation we describe our algorithms in terms of three-element tuples i, j, k, however, discussion generalizes to arbitrary loop nests.

6 BACKEND-INDEPENDENT VECTORIZATION

This section describes our vectorization algorithm. While vectorization is a longstanding problem and we build upon classical work on scalar expansion and loop vectorization [4], our algorithm is unique as it works on the MPC-IR SSA representation. We posit that vectorization over MPC-IR is a problem that warrants a fresh look, in part because of MPC's unique linear structure and in part because vectorization interacts with other MPC-specific optimizations in non-trivial ways (other work has explored manual vectorization and protocol mixing in an ad-hoc way, e.g., [25, 17, 34]).

6.1 Dependence Analysis

We build a dependence graph where the nodes are the MPC-IR statements and the edges represent the def-use relations. Since MPC-IR is an SSA form, *def-use edges* $X \rightarrow Y$ are explicit. We distinguish between *forward* edges where X appears before Y in the linear MPC-IR and *backward* edges where Y appears before X. Def-use edges are as follows:

- same-level edge $X \rightarrow Y$ where X and Y are in the same loop nest, say i, j, k. E.g., the def-use edge 9 to 10 in the Biometric MPC-IR in Listing 1(b) is a same-level edge. A same-level edge can be a backward edge in which case the target is a PHI-node. E.g., 15 to 4 in Biometric is a same-level backward edge.
- outer-to-inner $X \rightarrow Y$ where X is in an outer loop nest, say i, and Y is in an inner one, say j, k. E.g., 1 to 4 in Biometric is an outer-to-inner edge.
- inner-to-outer $X \rightarrow Y$ where X is a PHI-node in an inner loop nest, k, and Y is in the enclosing loop nest i, j. E.g., the def-use from 8 to 12 gives rise to an inner-to-outer edge. An inner-to-outer edge can be a backward edge as well, in which case both X and Y are PHI-nodes with the source X in a loop nested into Y's loop (not necessarily immediately).
- mixed forward edge $X \rightarrow Y$. X is in some loop i, j, k and Y is in a loop nested into i, j, k'. We transform mixed forward edges as follows. Let x be the variable defined at X. We add a variable and assignment $\mathsf{x}' = \mathsf{x}$ immediately after the i, j, k loop. Then we replace the use of x at Y with x'. This transforms a mixed forward edge into an "inner-to-outer" forward edge followed by an outer-to-inner forward edge. Thus, Basic Vectorization handles one of "same-level", "inner-to-outer", or "outer-to-inner" def-use edges.

We define *closure*(n) where n is a PHI-node as the set of nodes (i.e., statements) that form a same-level dependence cycle with n. The closure of n is defined as follows:

- n is in *closure*(n)
- X is in *closure*(n) if there is a same-level path from n to X, and $X \rightarrow n$ is a same-level back-edge.
- Y is in *closure*(n) if there is a same-level path from n to Y and there is a same-level path from Y to some X in *closure*(n).

6.2 Scalar and Array Expansion

Scalar and array expansion to the corresponding loop dimensionality exposes opportunities for vectorization. In the Biometric example, `d=S[i*D+j]-C[j]` (equiv. to `d=S[i,j]-C[j]`) gives rise to $N * D$ subtraction operations in the sequential schedule. It can be expanded. The argument arrays S and C and the scalar d are expanded and the statement becomes `d[i,j]=S[i,j]-C[i,j]`. The algorithm then detects that the statement can be vectorized.

The *raise_dim* function expands a scalar (or array). There are two versions of *raise_dim*. One reshapes an arbitrary access into a canonical read access in the corresponding loop. It takes the original array, the original access pattern function $f(i, j, k)$ in loop nest i, j, k and the loop bounds $((i{:}I), (j{:}J), (k{:}K))$ (cf. 5.2):

$$raise_dim(\mathsf{A}, f(i, j, k), ((i{:}I), (j{:}J), (k{:}K)))$$

It produces a new 3-dimensional array A' by iterating over i, j, k and setting each element of A' as follows:

$$\mathsf{A}'[i, j, k] = \mathsf{A}[f(i, j, k)]$$

The end result is that uses of $\mathsf{A}[f(i, j, k)]$ in loop nest i, j, k are replaced with canonical read accesses to $\mathsf{A}'[i, j, k]$ that can be vectorized. In the running Biometric example, $raise_dim(\mathsf{C}, j, (i{:}N, j{:}D))$ expands the 1-dimensional array C into a 2-dimensional array. The i, j loop now accesses the expanded array in the canonical way.

The other version of *raise_dim* lifts a lower-dimension expanded scalar (or array) into a higher-dimension for access in a nested loop. It is necessary when processing outer-to-inner dependences. Here A is a one dimensional array in loop nest i and raise dimension adds two additional dimensions; this version reduces to the above version by adding the access pattern function, which is just i:

$$raise_dim(\mathsf{A}, i, (i:I, j:J, k:K))$$

drop_dim is carried out when an expanded scalar (or array) written in an inner loop is used in an enclosing loop. It takes a higher-dimension array, say i, j, k and removes trailing dimensions, say j, k:

$$drop_dim(\mathsf{A}, (j:J, k:K))$$

It iterates over i and takes the result at the maximal index of j and k. This is the result at the last iterations of inner loop j:

$$\mathsf{A}'[i] = \mathsf{A}[i, J-1, K-1]$$

Conceptually, we treat all variables as arrays. There are three kinds of arrays.

- Scalars: We expand scalars into arrays for the purposes of vectorization. For those, all writes are canonical writes and all reads are canonical reads. We *raise dimension* when a scalar gives rise to an outer-to-inner dependence edge (e.g., `sum!2` in line 6 of the MPC-IR code will be raised to a 2-dimensional array since `sum!2` is used in the inner j-loop). We *drop dimension* when a scalar gives rise to an inner-to-outer dependence edge (e.g., `sum!3` for which the lifted inner loop computes D values, but the outer loop only needs the last one).
- Read-only input arrays: There are no writes, while we may have non-canonical reads, $f(i, j, k)$. Vectorization adds raise dimension operations at the beginning of the function to lift these arrays to the dimensionality of the loop where they are used, possibly *reshaping* the arrays.
- Read-write arrays: Writes are canonical (by restriction) but reads can be non-canonical. We may apply both raise and drop dimension, however, they respect the fixed dimensionality of the array. The array cannot be raised to a dimension lower than its canonical (fixed) dimensionality and it cannot be dropped to lower dimension. The restriction to canonical writes essentially reduces the case of arrays to the case of scalars, simplifying vectorization and correctnss reasoning.

6.3 Basic Vectorization Algorithm

There are two key phases of the algorithm. Phase 1 inserts raise dimension and drop dimension operations according to def-uses. E.g., if there is an outer-to-inner dependence, it inserts *raise_dim*, and if there is an inner-to-outer dependence, it inserts *drop_dim*. After this phase operations work on arrays of the corresponding loop dimensionality and we optimistically vectorize all arrays.

Phase 2 proceeds from the inner-most towards the outer-most loop. For each loop it anchors dependence cycles (closures) around pseudo PHI-nodes then removes vectorization from the dimension of that loop. There are two important points in this phase. First, it may break a loop into smaller loops which could allow vectorization in intermediate statements in the loop. Second, it creates opportunities for vectorization in the presence of write arrays, even though Cytron's SSA adds a backward edge to the array PHI-node and a straight-forward analysis would prohibit vectorization of all statements that access the array.

The code in blue color in the algorithm below highlights the extension with array writes. We advise the reader to omit the extension for now and consider just read-only arrays. We explain the extension in §6.5. (As many of our benchmarks include write arrays, it plays an important role.)

The algorithm includes a Phases 3, which reduces unnecessary dimensionality. This is an optional phase and our current implementation does not include it; thus, we elide it from this presentation.

{ Phase 1: Raise/drop dimension of scalars to corresponding loop nest. We traverse stmts linearly in MPC-IR. }
for each MPC *stmt*: $\mathsf{x} = Op(\mathsf{y}_1, \mathsf{y}_2)$ in loop i, j, k **do**
  for each argument y_n **do**
    case edge *stmt'* (def of y_n) → *stmt*(def of x) of
      same-level: y'_n is y_n
      outer-to-inner: add $\mathsf{y}'_n[i, j, k] = raise_dim(\mathsf{y}_n)$ at *stmt'* (more precisely, right after *stmt'*)
      inner-to-outer: add $\mathsf{y}'_n[i, j, k] = drop_dim(\mathsf{y}_n)$ at *stmt* (more precisely, in loop of *stmt* right after loop of *stmt'*)
  end for
  { Optimistically vectorize. I is vectorized dimension. }
  change *stmt* to $\mathsf{x}[I, J, K] = Op(\mathsf{y}'_1[I, J, K], \mathsf{y}'_2[I, J, K])$
end for
{ Phase 2: Recreating for-loops for cycles; *vectorizable* statements remain outside of for-loops. }
for each dimension d from highest to 0 **do**
  for each PHI-node n in loop $i_1, ..., i_d$ **do**
    compute *closure*(n)
  end for
  { closures cl_1 and cl_2 intersect if they have common statement or update same array; intersect of closures can be expanded or narrowed }
  while there are closure cl_1 and cl_2 that intersect **do**
    merge cl_1 and cl_2
  end while
  for each closure *cl* (after merge) **do**
    create `for` i_d `in` ... loop
    add PHI-nodes in *cl* to header block
    add target-less PHI-node for A if *cl* updates array A
    add statements in *cl* to loop in order of dependences
    { Dimension is not vectorizable: }
    change I_d to i_d in all statements in loop
    treat `for`-loop as one node for def-uses and closures: e.g., some def-use edges become same-level.
  end for
  for each target-less PHI-node A_1 = PHI(A_0,A_k) **do**
    in vectorizable stmts, replace use of A_1 with A_0
    discard PHI-node if not used in any *cl*, replacing A_1 with A_0 or A_k as necessary
  end for
end for
{ Phase 3: Remove unnecessary dimensionality.}

Consider our running example in Listing 1(b). Phase 1 will raise dimensions of `min_sum!1` to a 1-dimensional array as it is defined

	Syntax	Semantics	
s	$::= s_1; s_2$	$\gamma(s) = \gamma(s_1)\ ;\ \gamma(s_2)$	*sequence*
	$\mid \mathsf{x}[i,J,k] = \mathsf{op_SIMD}(\mathsf{y}_1[i,J,k], \mathsf{y}_2[i,J,k])$	$\gamma(\mathsf{x}[i,J,k] = \mathsf{op_SIMD}(\mathsf{y}_1[i,J,k], \mathsf{y}_2[i,J,k])) =$ $\mathsf{x}[i,0,k] = \mathsf{y}_1[i,0,k]\ \mathsf{op}\ \mathsf{y}_2[i,0,k]\ \ \mid\mid$ $\mathsf{x}[i,1,k] = \mathsf{y}_1[i,1,k]\ \mathsf{op}\ \mathsf{y}_2[i,1,k]\ \ \mid\mid \ldots \mid\mid$ $\mathsf{x}[i,J-1,k] = \mathsf{y}_1[i,J-1,k]\ \mathsf{op}\ \mathsf{y}_2[i,J-1,k]$	*operation*
	$\mid \mathsf{x}[i,J,k] = \mathsf{const}$	analogous	*constant*
	$\mid \mathsf{x}[i,J,k] = \mathsf{PHI}(\mathsf{x}_1[i,J,k], \mathsf{x}_2[i,J,k-1])$		*pseudo PHI*
	$\mid \mathsf{x}[i,J,k] = \mathsf{raise_dim}(\mathsf{x}'[i], (J{:}J, k{:}K))$		*raise dimension(s)*
	$\mid \mathsf{x}[i,J] = \mathsf{drop_dim}(\mathsf{x}'[i,J,k], k)$		*drop dimension(s)*
	$\mid$ for i in range(I) : s	$\gamma($for i in range(I) : $s) =$ $\gamma(s)[0/i]\ ;\ \gamma(s)[1/i]\ ;\ \ldots\ ;\ \gamma(s)[I-1/i]$	*loop*

Figure 3: MPC-IR Syntax and Semantics. γ defines the semantics of MPC-IR which is a linearization of input MPC-IR. A SIMD operation parallelizes operations across the vectorized J dimension. || denotes parallel execution, which is standard. γ of a for loop unrolls the loop. ; denotes sequential execution. Iterative MPC-IR trivially extends to non-vectorized dimensions over the enclosing loops.

outside of the loop but is used inside the i-loop. It will expand `C` into a 2-dimensional (i, j)-array. Phase 1 will also add *drop_dim* to drop the dimension of `sum!3`, which is defined in the inner loop and is of dimension (i, j), but is used in the outer i-loop and needs to align to that loop dimensionality.

Phase 2 starts with the inner j-loop. There are no dependences for the SUB and MUL statements (lines 9-10 in Listing 1(b)) and they are moved outside of the loop. The ADD is part of a cycle and it remains enclosed in a j-loop. Moving up to the outer i-loop, the addition j-loop is not part of a cycle in i and Phase 2 moves that loop outside vectorizing the i dimension of the summation (this results in the loop in lines 12-16 in Listing 1(c)). The MUX computations are part of cycles and they remain in i-loops.

6.4 Correctness Argument

We build a correctness argument as follows. First, we define the MPC-IR syntax. We then define the *linearization* of an MPC-IR program as an *interpretation* over the syntax. The linearization is a *schedule* as shown in §4. We prove a theorem that states that the Basic vectorization algorithm preserves the def-use relations, or in other words, linearization of the vectorized MPC-IR program gives rise to the exact same set of def-use pairs as linearization of the original program does. It follows easily that the schedule corresponding to the vectorized program computes the same result as the schedule corresponding to the original program.

MPC-IR Syntax. Fig. 3 states the syntax and linearization semantics of MPC-IR. Although notation is heavy, the linearization simply produces schedules as described in §2 and §4. The iterative MPC-IR (i.e., one without vectorized dimensions) gives rise to what we called sequential schedule where loops are unrolled and MPC-IR with vectorized dimensions gives rise to what we called parallel schedule. For simplicity, we consider only scalars and read-only arrays, however, the treatment extends to write arrays as well (with our restriction on array writes to canonical writes). $\mathsf{x}[i,J,k]$ denotes scalar variable x at loop nest i, j, k. (We use 3 dimensions, i, j, k, however, the treatment generalizes to arbitrary dimensions.) Upper case J denotes a vectorized dimension and lower case i, k denote iterative dimensions. There are two key semantic restrictions over the MPC-IR syntax: (1) x is a 3-dimensional array and (2) $\mathsf{x}[i,J,k]$ is enclosed in for-loops on non-vectorized dimensions i and k:

```
1 for i in range(I):
2     ...
3     for k in range(K):
4         ... x[i,J,k] ...
```

Linearization. The linearization function γ is defined as an interpretation over MPC-IR syntax, which is standard. It is shown in the middle column of Fig. 3. The linearization of an `op_SIMD` statement expands the vectorized dimension(s) into *parallel* statements; || introduces SIMD (parallel) execution. The linearization of the for i in range(I) : s statement simply unrolls the loop substituting i with 0, 1, etc.; here ; denotes *sequential* execution.

As an example, consider the vectorized MPC-IR from our running example. All variables are two dimensional arrays and the loop is vectorized in I but iterative in j:

```
1 for j in range(0, D):
2   sum!3[I,j] = PHI(sum!2[I,j], sum!4[I,j−1])
3   sum!4[I,j] = ADD(sum!3[I,j], p[I,j])
```

Assuming $D = 2$ and $I = 2$ for simplicity, linearization produces the following schedule:

```
1 sum!3[0,0] = PHI(sum!2[0,0], sum!4[0,−1]) || sum!3[1,0] = PHI(sum!2[1,0], sum!4[1,−1])
2 ;
3 sum!4[0,0] = ADD(sum!3[0,0], p[0,0]) || sum!4[1,0] = ADD(sum!3[1,0], p[1,0])
4 ;
5 sum!3[0,1] = PHI(sum!2[0,1], sum!4[0,0]) || sum!3[1,1] = PHI(sum!2[1,1], sum!4[1,0])
6 ;
7 sum!4[0,1] = ADD(sum!3[0,1], p[0,1]) || sum!4[1,1] = ADD(sum!3[1,1], p[1,1])
```

Note that by definition of the pseudo PHI function, `PHI(sum!2[0,0], sum!4[0,-1])` evaluates to `sum!2[0,0]` and therefore, the -1 index in the second argument does not matter.

Statements and def-uses over MPC-IR. Let a be an MPC-IR program. Since MPC-IR is an SSA form, def-use edges in a are explicit (as in §6.1): if $s_0 \in a$ defines variable x, e.g., x = ..., $s_1 \in a$ uses x, e.g., ... = ...x, then there is a def-use edge from s_0 to s_2. We write $s_0[i, j, k]$ for statement s_0 enclosed in loop nest i, j, k.

Let a_0, a_1 be two MPC-IR programs. Two statements, $s_0 \in a_0$ and $s_1 \in a_1$ are *same*, written $s_0 \equiv s_1$ if they are of the same operation and they operate on the same variables: same variable name and same dimensionality. Recall that dimensions in MPC-IR are either iterative, lower case, or vectorized, upper case. Two statements are same even if one operates on an iterative dimension and the other one operates on a vectorized one, e.g., $s_0[i, j, k] \equiv s_1[I, j, K]$.

Statements and def-uses over linearized schedule. An *atomic* statement is a statement produced by linearization. We write $\underline{s_0}$ to denote statements in the schedule as well as $\underline{s_0}[\underline{i}, \underline{j}, \underline{k}]$ to denote fully instantiated values of i, j, and k, such as for example $\underline{s_0}[0, 1, 0]$. Clearly, the linearization of same statements produces the same set of atomic statements in the linearized schedule.

A def-use pair of atomic statements, denoted $\underline{s_0} \rightarrow \underline{s_1}$ (indexing implicit), is defined in the standard way as well: $\underline{s_0}$ writes a location, say $\mathsf{x}[\underline{i}, \underline{j}, \underline{k}]$, and $\underline{s_1}$ reads the same location.

Formal treatment. Property P defined below relates the linearized schedule of iterative MPC-IR program a_0 to the linearized schedule of the vectorized program a_1. More precisely, a_0 is the MPC-IR program augmented with raise and drop dimension statements, i.e., Phase 1 without optimistic vectorization of all dimensions. a_1 is produced from a_0 by Phase 2 of the Basic vectorization algorithm. For clarity, we elide raise and drop dimension statements; extending def-use reasoning is straightforward.

Definition 1. *We say that $\gamma(a_0) \equiv \gamma(a_1)$ iff (1) atomic statement $\underline{s}[\underline{i}, \underline{j}, \underline{k}] \in \gamma(a_0) \Leftrightarrow \underline{s}[\underline{i}, \underline{j}, \underline{k}] \in \gamma(a_1)$ and (2) $\underline{s_0} \rightarrow \underline{s_1} \in \gamma(a_0) \Leftrightarrow \underline{s_0} \rightarrow \underline{s_1} \in \gamma(a_1)$ (indexing implicit).*

Let us first prove the following lemma which states that Basic vectorization preserves statements and def-use edges in the original MPC-IR.

Lemma 1. *For each statement s in a_0, there is same statement s' in a_1, and vice versa. For each def-use edge e in a_0, there is a same edge e' in a_1, and vice versa.*

Proof. Proof sketch of Lemma 1. Phase 2 of Basic Vectorization does not introduce any new statements in the code, it just vectorizes dimensions. Similarly, reordering of statements preserves exactly the def-use edges in the original MPC-IR. □

The main theorem below states that Basic vectorization preserves def-use edges.

Theorem 1. *$\gamma(a_0) \equiv \gamma(a_1)$.*

Proof. Proof sketch of Theorem 1. The first condition of property P follows directly from Lemma 1. The proof of the second condition is by analysis of the def-use edges in $\gamma(a_0)$ and the corresponding edges in $\gamma(a_1)$; the key is that Basic vectorization preserves the def-uses in a_0.

A forward edge $s_0 \rightarrow s_1 \in a_0$ remains a forward edge in a_1. Without loss of generality, let us assume an outer loop i and a nested loop j. The forward edge entails the following ordering in linearization $\gamma(a_0)$:

$$\begin{array}{ll} \underline{s_0}[\underline{i}] \; ; \; \underline{s_1}[\underline{i}, \underline{j}] & \text{outer-to-inner edge} \\ \underline{s_0}[\underline{i}, \underline{j}] \; ; \; \underline{s_1}[\underline{i}, \underline{j}] & \text{same-level edge} \\ \underline{s_0}[\underline{i}, \underline{j}] \; ; \; \underline{s_1}[\underline{i}] & \text{inner-to-outer edge} \end{array}$$

meaning that for a fixed $\underline{i}$, def $\underline{s_0}[\underline{i}]$ is scheduled *before* use $\underline{s_1}[\underline{i}]$. Due to the preservation of the edge in a_1, the above ordering holds in $\gamma(a_1)$ as well.

Consider a backward edge $s_0 \rightarrow s_1 \in a_0$. We have that s_1 is a PHI-node in some loop, say i. There are two cases: (1) there is a path of forward edges from s_1 to s_0, and (2) there is no such path. In case (1) Basic vectorization detects a cycle (closure) around s_1, and therefore, $s_0 \rightarrow s_1$ remains a backward edge in a_1. The linearization of the backward edge imposes ordering $\underline{s_0}[\underline{i} - 1] \; ; \; \underline{s_1}[\underline{i}]$ and due to preservation of the backward edge in a_1, the ordering holds in $\gamma(a_1)$ as well. In case (2) Basic vectorization may turn the backward edge into a forward one, however, it preserves the $\underline{s_0}[\underline{i} - 1] \; ; \; \underline{s_1}[\underline{i}]$ ordering constraint by construction. □

The key argument is that the Basic vectorization algorithm preserves def-uses when it transforms a_0 into a_1. This leads to preservation of concrete edges in $\gamma(a_0)$ into $\gamma(a_1)$. A corollary of the main theorem follows:

Corollary 1.1. $\gamma(a_0)$ and $\gamma(a_1)$ produce same result, or more precisely, for every location $\mathsf{x}[\underline{i}, \underline{j}, \underline{k}]$, $\gamma(a_0)$ and $\gamma(a_1)$ compute the same result.

Proof. Proof sketch of Corollary 1.1. This can be established by induction over the length of def-use chains of computation in $\gamma(a_0)$. Assume that for all chains of length $\leq n$ all locations $l[\underline{i}, \underline{j}, \underline{k}]$ hold the same value in $\gamma(a_0)$ and $\gamma(a_1)$. A chain of length $n + 1$ results from the execution of a statement $\mathsf{x}[\underline{i}, \underline{j}, \underline{k}] = \mathsf{y}[\underline{i}, \underline{j}, \underline{k}] \; op \; \mathsf{z}[\underline{i}, \underline{j}, \underline{k}]$. By property P, there is the same statement in $\gamma(a_1)$ and it is scheduled after the definitions of $\mathsf{y}[\underline{i}, \underline{j}, \underline{k}]$ and $\mathsf{z}[\underline{i}, \underline{j}, \underline{k}]$. By the inductive hypothesis $\mathsf{y}[\underline{i}, \underline{j}, \underline{k}]$ and $\mathsf{z}[\underline{i}, \underline{j}, \underline{k}]$ hold the same values in $\gamma(a_0)$ as in $\gamma(a_1)$. Therefore, locations $\mathsf{x}[\underline{i}, \underline{j}, \underline{k}]$ hold the same value as well. We remark that due to the SSA form, each location $l[\underline{i}, \underline{j}, \underline{k}]$ is defined at most once. For clarity, we elide PHI nodes and raising and dropping dimensions; extending def-use reasoning is straight forward. □

6.5 Extension with Array Writes

Array writes may introduce infeasible loop-carried dependencies. Consider an example from [3]:

```
1 for i in range(N):
2   A[i] = B[i] + 10;
3   B[i] = A[i] * D[i-1];
4   C[i] = A[i] * D[i-1];
5   D[i] = B[i] * C[i];
```

In Cytron's SSA this code (roughly) translates into

```
1 for i in range(N):
2   A_1 = PHI(A_0,A_2)
3   B_1 = PHI(B_0,B_2)
4   C_1 = PHI(C_0,C_2)
5   D_1 = PHI(D_0,D_2)
6   A_2 = update(A_1, i, B_1[i] + 10);
7   B_2 = update(B_1, i, A_2[i] * D_1[i-1]);
8   C_2 = update(C_1, i, A_2[i] * D_1[i-1]);
9   D_2 = update(D_1, i, B_2[i] * C_2[i]);
```

`B_1=PHI(B_0,B_2)` anchors a cycle that includes statement `A_2=update(A_1,i,B_1[i]+10)`; a naive approach will not vectorize the latter statement even though there is no loop-carried dependency from the write of `B_1[i]` at 7 to the read of `...=B_1[i]` at 6.

We remove certain infeasible loop-carried dependencies that are due to array writes. Consider the "middle" loop j in loop nest i, j, k. Only if an update (definition) $\mathsf{A}_m[f(i, j, k)] = \ldots$ at some iteration $\underline{j}$ references the *same* array element as a use $\ldots = \mathsf{A}_n[f'(i, j, k)]$ at some later iteration $\underline{j}'$, we may have a loop-carried dependence for A due to this def-use pair. In contrast, standard SSA inserts a loop-carried dependency every time there is an array update. The

algorithm examines all def-use pairs in loop j, including defs and uses in nested loops, searching for values $\underline{i}, \underline{j}, \underline{j}', \underline{k}, \underline{k}'$ that satisfy $f(\underline{i}, \underline{j}, \underline{k}) = f'(\underline{i}, \underline{j}', \underline{k}')$. If such values exist for some def-use pair, then there is a potential loop-carried dependence on A; otherwise there is not and we can remove the spurious backward edge thus "freeing up" statements for vectorization. The algorithm below reasons about loop-carried dependencies from updates to array A to uses of A in loop j:

for each array A written in loop j **do**
  { including enclosed loops in j }
  dep = False
  for each def-of-A: $A_m[f(i, j, k)] = \ldots$ and
  use-of-A: $\ldots = A_n[f'(i, j, k)]$ in loop j **do**
    if $\exists \underline{i}, \underline{j}, \underline{j}', \underline{k}, \underline{k}'$, s.t. $0 \leq \underline{i} < I$, $0 \leq \underline{j}, \underline{j}' < J$, $0 \leq \underline{k}, \underline{k}' < K$, $\underline{j} < \underline{j}'$,
    and $f(\underline{i}, \underline{j}, \underline{k}) = f'(\underline{i}, \underline{j}', \underline{k}')$ **then**
      dep = True
    end if
  end for
  if dep == False **then**
    remove back edge into A's PHI-node in loop j.
  end if
end for

We use Z3 [26] to check satisfiability of the formula

$$(0 \leq \underline{i} < I) \wedge (0 \leq \underline{j}, \underline{j}' < J) \wedge (0 \leq \underline{k}, \underline{k}' < K) \wedge (\underline{j} < \underline{j}') \wedge f(\underline{i}, \underline{j}, \underline{k}) = f'(\underline{i}, \underline{j}', \underline{k}')$$

Formulas f and f' are simple as loop nests are typically of depth 2-3. Therefore, Z3 completes the process instantly.

Consider the earlier example. There is a single loop, i. Clearly, there is no pair $\underline{i}$ and $\underline{i}'$, where $\underline{i} < \underline{i}'$ that make $\underline{i} = \underline{i}'$ in either def-use pair 6-7 or 6-8. Therefore, we remove the backward edge from 6 to the phi-node 2. Analogously, we remove the backward edges from 7 to 3 and from 8 to 4. However, there are many values $\underline{i} < \underline{i'}$ that make $\underline{i} = \underline{i'} - 1$ and the backward edge from 9 to 5 remains (def-use pairs for D). As a result of removing these spurious edges, Vectorization will find that statement 6 is vectorizable. Statements 7, 8 and 9 will correctly appear in the reconstructed for-loop.

This step renders some array PHI-nodes *target-less*. We handle target-less PHI-nodes with a minor extension of Basic vectorization (Phase 2, extension shown in blue). First, we merge closures that update the same array. This simplifies handling of array PHI-nodes: if each closure is turned into a separate loop, each loop will need to have its own array PHI-node to account for the update and this would complicate the analysis. Second, we add the target-less node of array A back to the closure that updates A — the intuition is, even if there is no loop-carried dependence from writes to reads on A, A is written and the write (i.e., update) cannot be vectorized due to a different cycle; therefore, the updated array has to carry to the next iteration of the loop. Third, when the PHI-node remains target-less, i.e., cases when the array write can be vectorized, we have to properly remove the PHI-node replacing uses of the left-hand side of the PHI-node with its arguments (last snippet in blue).

Recall that we restrict array updates to *canonical updates*, i.e., an update $A[i, j] = \ldots$ is enclosed in loops on i and j. It may be enclosed into a nested loop i, j, k, however, the indices correspond to the outermost loops. This restriction ensures that the array shape does not change and raise dimension and drop dimension can be applied in the same away as in the basic case, thus allowing us to extend correctness reasoning from the basic case. We will look to relax the restriction in future work, when additional optimizations are overlaid to vectorizåtion (*cf.* §9).

7 COMPILER BACKENDS

Translation into the MOTION and MP-SPDZ backends is done by standard interpretation (i.e., syntax-directed translation) over the MPC-IR syntax shown in Fig. 3.

Naturally, translation into C++ code for the MOTION framework presents more challenges. First, MPC-IR requires `shared` qualifiers only on input variables, while MOTION requires all variables to be typed either `shared` or `plain`. To resolve this, we infer qualifiers for all variables by performing *taint analysis* (see [39]). Another challenge is dealing with public values, e.g., constants. Since there is no support for these, one has to provide such public values as shared input from one of the parties. Input gates are expensive and a naive implementation could introduce a performance hit. Instead, we keep shared copies of plain (public) variables, and update them in lock-step with updates to plain variable. When a plain variable is needed in a shared context, we use the shared copy. Details on the translation can be found in [39].

The current implementation supports unsigned integers, booleans, tuples, and lists at the frontend level, and it is our intention to extend it with additional types. At the backend level it supports unsigned integers and booleans for secure computation. The MPC protocol parameters e.g., computational/statistical security parameters, are specified independently for each backend. In our evaluation, we use each backend's default values for them.

8 EVALUATION AND ANALYSIS

We evaluate on the two major MPC backend frameworks, MOTION (commit `6a7c1c7`) and MP-SPDZ (commit `640b1a9`). Specifically, we compare the iterative (non-vectorized) version of a benchmark against the vectorized one. We compare on a benchmark suite that includes standard MPC benchmarks [17, 34] as well as new ones from [30]. Details on the suite can be found in the full paper [39]. Note that since we work at the higher level of MPC-IR, the source-to-source compilation overhead of our compiler is negligible: the slowest benchmark to compile to either MP-SPDZ or MOTION takes ≈ 65ms (average of 100 runs).

8.1 MOTION Experiments

8.1.1 Experiment Setup. Experiment hardware is generously provided by CloudLab [27]. For the network, we consider two settings, LAN and WAN. In the LAN setting, we use `c6525-25g` machines connected via a 10Gbps link with < 1ms latency. These machines are equipped with 16-core AMD 7302P 3.0GHz processors and 128GB of RAM. For WAN, we use a `c6525-25g` machine (located in Utah, US) and a `c220g1` machine (located in Wisconsin, US). The `c220g1` machine is equipped with two Intel E5-2630 8-core 2.40GHz processors and 128GB of RAM. We measured the connection bandwidth between these machines to be 560Mbps and average round trip time (RTT) to be 38ms.

We run 2PC and 3PC experiments on LAN with input datasets that range in size from 2 to 4096. In the WAN setting we only perform 2PC experiments to save time; as evidenced by LAN experiments in §8.1.2, 3PC experiments would only take longer to run. We run all experiments 5 times and report average values of various metrics. The standard deviation, shown as error bar on top of the histogram bars in the graphs, is at most 4.5% of the mean. Therefore, more runs will not significantly improve result accuracy. Due to space constraints, we present a subset of all results; the full set can be found in [39]. Tables and graphs in this section are with the largest dataset for which the non-vectorized run completes (it typically runs out of memory while the vectorized runs continue).

8.1.2 Results and Analysis. A summary of the effects of vectorization (MOTION backend) is presented in Table 2. In addition we show graphically circuit evaluation times in Fig. 4 and communication size in Fig. 5. In terms of amenability to vectorization, we divide benchmarks into 3 categories: (1) *High:* these include Convex Hull, Cryptonets (Max Pooling), Minimal Points and Private Set Intersection. These benchmarks are highly parallelizable and see 47x to 23x speedup in BMR, and 33x to 23x in GMW. (2) *Medium:* these include Biometric Matching, DB Variance, Histogram, Inner product, K-means iteration and MNIST ReLU. These benchmarks have non-parallelizable phases e.g. the summing phase of inner product and Biometric Matching. [4] Still, most computation is parallelizable and it results in speedup from 23x to 5x in BMR, and 23x to 3x in GMW protocol. (3) *Low:* these include the Database Join and the regular expression benchmarks (Count 102, Count 10, Longest 102 and Max Distance between Symbols). Fewer operations in these programs are parallelizable, thus the speedup is lower. We see a speedup from 2x to 1.1x in BMR. In GMW, Database Join, Count 102 and Count 10s see speedup from 1.3x to 1.1x. However, Longest 102 and Max distance suffer a slowdown of 0.5x. There is opportunity for vectorization in these benchmarks according to our analytical model, particularly, there is a large EQ operation that is vectorized, although a large portion of the loop cannot be vectorized. We observe that transformation to vectorized code increases multiplicative depth and, the negative effect of increased depth is more noticeable in a round-based protocol like GMW. We conjecture that MOTION performs optimizations over the non-vectorized loop body that decreases the depth; also, EQ is relatively inexpensive in Boolean GMW and BMR compared to ADD and MUL, which also de-emphasizes the benefit of vectorization. We propose a simple heuristic (although we do leave all the benchmarks in the table): if the transformation increases circuit depth beyond some threshold (e.g. more than 10% of the original circuit), we can reject the transformation. In some settings it may still be desirable to vectorize e.g. in data constrained environments, as vectorization reduces communication costs.

As shown in Fig. 5, vectorization reduces communication, up to 12x in GMW and 3x in BMR (see [39] for details). In summary, vectorization enables better packing and it impacts an interactive protocol like GMW more than a constant round protocol like BMR.

[4]We remark that summation can be parallelized as a $log(N)$-depth tree (e.g., as in [16]). This is an instance of the divide-and-conquer paradigm [30] and we have chosen to leave it as future work — divide-and-conquer can be applied more generally and in a principled way in conjunction with vectorization, rather than just on summation.

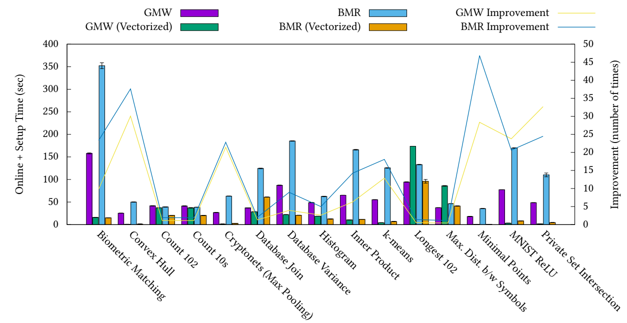

Figure 4: 2PC: Circuit evaluation time (Setup+Online) in seconds, LAN setting. Error bars show standard deviation.

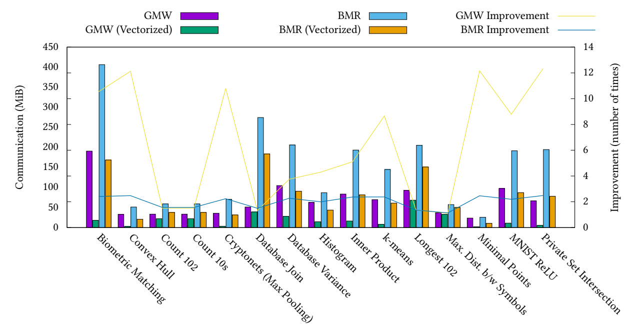

Figure 5: 2PC: Communication size.

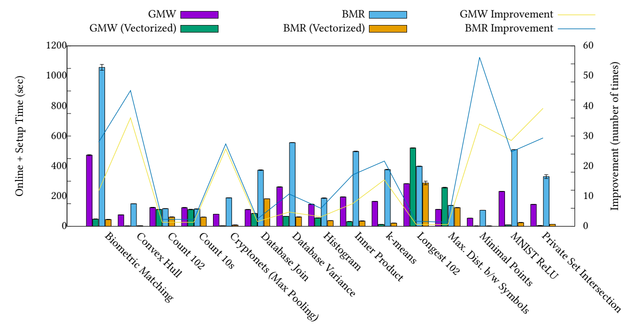

Figure 6: 3PC: Circuit evaluation time (Setup+Online) in seconds, LAN setting. The error bars show standard deviation.

In the 3PC setting, we observe that, as expected, evaluation time (Fig. 6) is higher than in the 2PC setting. This is a direct consequence of higher online time for the GMW protocol. While online time for BMR remains roughly the same (which is expected because the online phase is essentially local in BMR), it suffers a slowdown in the setup phase; the circuit for 3 parties requires more computation. The experiments for 3PC confirm that adding more parties to an MPC increases resource requirements. More details are available in the full version of the paper [39].

Table 2: Non-vectorized vs. vectorized comparison in 2PC LAN setting. Times in seconds, Communication in MiB, Numbers in 1000s and rounded to nearest integer; vectorized benchmarks have (V) in name. All metrics are produced by MOTION.

	GMW						BMR					
Benchmark	Online	Setup	# Gates	Circ Gen	# Msgs	Comm.	Online	Setup	# Gates	Circ Gen	# Msgs	Comm.
Biometric Matching	146	16	1,784	119	1,413	140	89	263	1,595	139	2,716	312
Biometric Matching (V)	12	4	34	2	28	14	2	13	30	4	61	130
Convex Hull	48	6	551	40	516	51	28	72	494	39	695	80
Convex Hull (V)	0	1	2	0	1	4	0	2	1	1	2	32
Count 102	79	6	418	35	525	52	15	62	269	33	785	92
Count 102 (V)	71	5	316	24	332	34	11	30	167	16	304	59
Count 10s	79	6	419	35	525	52	14	62	270	33	785	92
Count 10s (V)	71	4	316	24	332	34	11	29	167	16	304	59
Cryptonets (Max Pooling)	50	11	688	46	554	55	36	89	608	51	898	110
Cryptonets (Max Pooling) (V)	1	1	7	1	2	5	2	4	7	2	12	49
Database Join	70	8	433	48	790	80	19	229	458	119	3,518	427
Database Join (V)	54	6	320	35	575	61	16	112	320	57	1,457	285
Database Variance	166	18	2,009	135	1,639	163	95	269	1,708	145	2,795	320
Database Variance (V)	37	6	321	24	334	43	10	30	170	13	178	141
Histogram	94	10	862	68	979	97	27	94	491	51	1,132	135
Histogram (V)	33	5	166	16	164	23	7	17	92	13	154	68
Inner Product	127	15	1,675	108	1,308	130	83	250	1,526	134	2,623	301
Inner Product (V)	16	5	158	12	165	25	6	18	83	7	86	127
k-means	108	12	1,333	88	1,090	108	63	185	1,141	99	1,958	225
k-means (V)	6	3	47	4	43	12	2	11	32	4	54	95
Longest 102	93	7	650	52	713	71	26	93	475	49	1,091	128
Longest 102 (V)	169	6	544	41	519	53	25	60	369	33	605	95
Max. Dist. b/w Symbols	71	8	572	43	576	57	24	69	397	38	748	89
Max. Dist. b/w Symbols (V)	166	7	538	39	512	51	24	57	363	32	589	78
Minimal Points	35	5	458	31	369	37	24	46	401	26	347	40
Minimal Points (V)	0	1	1	0	1	3	0	1	1	0	1	16
MNIST ReLU	132	31	1,843	126	1,483	152	98	247	1,630	135	2,401	298
MNIST ReLU (V)	3	3	25	3	9	17	5	11	25	5	33	136
Private Set Intersection	95	9	558	59	1,049	104	22	186	591	96	2,639	302
Private Set Intersection (V)	1	2	1	2	1	8	1	8	2	4	2	122

8.2 MP-SPDZ Experiments

The MOTION compiler does not automatically apply vectorization; hence our MOTION experiments compare our vectorization optimization to "no optimization" (the iterative version of the benchmark) and, as expected, demonstrate high gains. Instead, MP-SPDZ does use vectorization but at a lower level than ours [36, Section 6.1] — the MP-SPDZ compiler unrolls the loop and merges (same) instructions from different iterations that may occur in parallel into a vector; the VM runs the vectorized instruction. Thus, it is reasonable to ask if our backend-independent optimization is compatible with MP-SPDZ and whether or not it still yields improvements. Our relevant experiments discussed below answer both these questions to the affirmative.

8.2.1 Experiment Setup. We run a representative subset of our MOTION experiments in the LAN setting (c.f. LAN configuration above). We remark that having already demonstrated the benefits of our optimization on a range of experiments with MOTION, the goal of our MP-SPDZ experiments is to address the above questions. Therefore, we chose a subset of benchmarks to run with MP-SPDZ that give a clear answer to the questions above while also demonstrating the delicate points of applying our optimizer on a back-end which already vectorizes (MP-SPDZ) vs. one that does not (MOTION). In particular, we focus on 2PC with semi-honest adversary for protocols following all three MPC paradigms (Boolean/Arithmetic GMW and BMR). As an extra point, to demonstrate that our optimization can also be applied to maliciously secure protocols, we run 2PC MP-SPDZ with the MASCOT protocol. Concretely, we run MP-SPDZ in the Arithmetic setting with the MASCOT and SOHO protocols, and in the Binary setting with the Semi-bin and Semi-BMR protocols. We run with datasets that trigger program complexity of at least $O(10^3)$. Our results are discussed below.

8.2.2 Results and Analysis. Table 3 summarizes our experiments. Vectorization has virtually no impact in the Arithmetic setting. In contrast, it has significant impact on compilation in the Binary setting, reducing compilation time from about 11% on Max Distance to 25x on Biometric. It has significant impact on memory footprint as well (22x in Biometric). We conjecture that the above discrepancy is due to the following reason: In the Arithmetic setting vectorization of data (typically integer arrays) through the API is as costly as merging of data and instructions by the MP-SPDZ compiler. Granularity of instructions is larger compared to the Binary setting, leading to fewer nodes and edges and lower complexity of the dependence graph and dependence analysis. In contrast, in the Binary setting's finer granularity of instructions (e.g., multiplication of two integers is expressed in terms of multiple AND and XOR instructions) leading to a larger and denser graph and significantly more complex dependence analysis. Performing dependence analysis apriori on the very short MPC-IR, which is what our vectorization analysis does, informs the MP-SPDZ compiler and reduces analysis time and memory footprint.

We note in passing, that our experiments demonstrate that vectorization slightly increases the number of rounds on the regular expression programs (e.g., Count 10s). This is consistent with the

Table 3: MP-SPDZ Compilation/execution stats, Security parameter = 40, Bit length = 32, 2PC 10Gbps LAN setting, Times in seconds, Memory/data in MBs, vectorized benchmarks have (V) in their name.

	Arithmetic								Binary							
					MASCOT		SOHO						Semi-bin		Semi-BMR	
Benchmark	Compilation (s)	Peak Mem.	# Int Triples	# VM Rounds	Run (s)	Data	Run (s)	Data	Compilation (s)	Peak Mem.	# Bit Triples	# VM Rounds	Run (s)	Data	Run (s)	Data
Biometric Matching	15.66	587.25	64,512	4,098	21.06	2,322.25	4.73	37.39	824.90	22,100.97	3,300,864	3,628	1.11	87.45	373.32	92,928.20
Biometric Matching (V)	16.16	605.53	64,512	4,098	21.15	2,322.25	4.72	37.39	**32.49**	**1,014.21**	3,300,864	3,651	0.67	41.73	369.92	92,928.20
Convex Hull	3.77	109.37	1,006,848	75	327.71	36,706.00	66.54	555.19	157.08	4,297.98	536,576	74	0.20	14.37	60.82	15,108.50
Convex Hull (V)	3.78	148.31	1,006,848	76	328.75	36,706.00	66.54	555.19	**18.64**	**612.45**	665,536	74	0.18	11.85	75.55	18,737.10
Count 10s	0.84	44.32	98,816	1,032	45.80	5,054.18	10.31	75.41	35.02	1,058.98	117,760	1,065	0.20	7.67	13.40	3,315.74
Count 10s (V)	0.94	51.88	98,816	1,545	45.82	5,054.18	10.27	75.41	**26.64**	**784.60**	117,760	2,086	0.24	6.91	13.44	3,325.75
Count 102	0.81	42.38	98,816	1,537	45.90	5,054.18	10.32	75.41	34.91	1,040.84	117,760	1,065	0.18	7.66	13.34	3,315.74
Count 102 (V)	0.94	51.36	98,816	15,37	45.86	5,054.18	10.33	75.41	**26.64**	**784.74**	117,760	2,087	0.24	6.91	13.75	3,315.74
Cryptonets (Max Pooling)	1.85	74.26	371,712	25	120.66	13,562.30	24.66	205.80	83.75	2,526.37	291,840	22	0.15	10.46	35.41	8,217.91
Cryptonets (Max Pooling) (V)	2.04	88.11	371,712	25	121.37	13,562.30	24.68	205.80	**33.01**	**1,140.89**	291,840	22	0.13	7.41	34.30	8,217.91
Database Join	8.62	244.82	786,432	12	365.42	40,296.70	78.17	558.06	176.04	5,812.58	774,144	10	0.29	23.97	93.54	21,797.80
Database Join (V)	9.56	281.16	786,621	12	365.34	40,316.20	78.36	561.57	**150.76**	**4,973.32**	774,237	10	0.27	22.47	91.91	21,800.90
Inner Product	0.14	24.18	512	2	0.20	19.54	0.47	5.51	170.52	4,852.62	669,696	551	0.31	22.27	76.38	18,855.90
Inner Product (V)	0.25	28.77	512	2	0.21	19.54	0.47	5.51	**19.74**	**620.45**	669,696	568	0.30	22.95	75.93	18,855.90
Longest 102	16.32	594.25	161,792	4,620	66.87	7,337.38	14.73	107.25	51.51	1,463.79	168,960	4,110	0.36	8.49	18.87	4,757.77
Longest 102 (V)	16.12	601.64	161,792	2,049	67.28	7,337.38	14.74	107.25	**39.16**	**1,159.94**	168,960	5,132	0.39	7.70	19.27	4,757.77
Max Dist. btw Symbols	15.40	579.75	94,720	4,105	36.33	3,961.44	7.84	59.57	41.07	1,206.88	131,584	3,600	0.33	7.89	14.89	3,706.72
Max Dist. btw Symbols (V)	15.21	573.94	94,720	4,616	36.02	3,961.44	7.97	59.57	**36.70**	**1,122.38**	131,584	3,600	0.30	7.64	15.00	3,706.72
Minimal Points	3.35	103.69	991,360	74	320.47	36,159.60	65.66	545.87	166.72	4,289.05	540,672	73	0.20	14.43	61.61	15,224.30
Minimal Points (V)	3.77	158.95	991,360	74	321.88	36,159.60	65.56	545.87	**21.21**	**621.64**	667,648	73	0.18	11.79	75.75	18,796.50
MNIST ReLU	6.84	212.63	991,232	9	321.28	36,159.70	65.95	546.00	266.69	7,632.11	778,240	8	0.34	24.07	99.70	21,916.20
MNIST ReLU (V)	6.08	233.02	991,232	9	320.73	36,159.70	65.47	546.00	**94.32**	**3,163.99**	778,240	8	0.26	15.94	101.66	21,916.20
Private Set Intersection	11.01	288.23	1,048,704	137	485.46	53,722.30	104.53	745.79	118.31	3,784.08	528,384	135	0.19	14.23	59.39	14,877.10
Private Set Intersection (V)	9.36	325.30	1,048,704	137	487.23	53,722.30	104.24	745.79	**23.28**	**757.01**	528,384	135	0.13	6.24	59.33	14,877.10

MOTION results where we observed little improvement or even slowdown. Indeed, our vectorization algorithm breaks the single loop into several loops separating smaller loops with a vectorized operation, typically equality; we conjecture that the vectorized operation serves as a barrier preventing the MP-SPDZ compiler from merging instructions it would otherwise merge in the non-vectorized single-loop program (resulting in improved round complexity). Nevertheless, our LAN experiments show that the above increase in rounds is not reflected as slowdown on running time, so the compilation savings make our optimization worthwhile.

9 CONCLUSION AND FUTURE WORK

We presented a formalization of the MPC-IR intermediate language followed by a specific backend-independent optimization at the level of MPC-IR: novel SIMD-vectorization. We demonstrated that vectorization has significant impact on performance and showcased the backend-independent nature of our optimization by applying it to two mainstream and parameterizable MPC frameworks, namely MOTION and MP-SPDZ.

We are excited about the opportunities for future work — integration with protocol mixing, divide-and-conquer reasoning and parallelization, as well as inter-procedural context-sensitive analysis at the level of MPC-IR will improve MPC programmability and efficiency.

10 ACKNOWLEDGEMENTS

We thank the reviewers for their constructive feedback. RPI authors are supported by NSF grants #1814898 and #2232061. Purdue authors are supported by the Algorand Centers of Excellence program managed by Algorand Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Algorand Foundation.

REFERENCES

[1] Cosku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. 2021. Viaduct: an extensible, optimizing compiler for secure distributed programs. In *ACM PLDI 2021*. Stephen N. Freund and Eran Yahav, (Eds.) ACM, (June 2021), 740–755.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools. Addison-Wesley series in computer science / World student series edition.* Addison-Wesley.

[3] Alexander Aiken and Alexandru Nicolau. 1988. Optimal loop parallelization. In *ACM PLDI 1988*. Richard L. Wexelblat, (Ed.) ACM, (June 1988), 308–317.

[4] Randy Allen and Ken Kennedy. 1987. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9, 4, 491–542.

[5] Gene M. Amdahl. 2013. Computer architecture and amdahl's law. *Computer*, 46, 12, 38–46.

[6] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida. 2018. Generalizing the SPDZ compiler for other protocols. In *ACM CCS 2018*. David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, (Eds.) ACM Press, (Oct. 2018), 880–895.

[7] Donald Beaver, Silvio Micali, and Phillip Rogaway. 1990. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*. ACM Press, (May 1990), 503–513. DOI: 10.1145/100216.100287.

[8] Assaf Ben-David, Noam Nisan, and Benny Pinkas. 2008. FairplayMP: a system for secure multi-party computation. In *ACM CCS 2008*. Peng Ning, Paul F. Syverson, and Somesh Jha, (Eds.) ACM Press, (Oct. 2008), 257–266.

[9] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*. ACM Press, (May 1988), 1–10.

[10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. 2010. The polyhedral model is more widely applicable than you think. In *Compiler Construction, CC 2010*. Rajiv Gupta, (Ed.) Vol. 6011. Springer, 283–303.

[11] Marina Blanton and Paolo Gasti. 2011. Secure and efficient protocols for iris and fingerprint identification. In *ESORICS 2011* (LNCS). Vijay Atluri and Claudia Díaz, (Eds.) Vol. 6879. Springer, Heidelberg, 190–209.

[12] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: a framework for fast privacy-preserving computations. In *ESORICS 2008* (LNCS). Sushil Jajodia and Javier López, (Eds.) Vol. 5283. Springer, Heidelberg, (Oct. 2008), 192–206.

[13] Peter Bogetoft et al. 2009. Secure multiparty computation goes live. In *FC 2009* (LNCS). Roger Dingledine and Philippe Golle, (Eds.) Vol. 5628. Springer, Heidelberg, (Feb. 2009), 325–343.

[14] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. 2022. Motion: a framework for mixed-protocol multi-party computation. *ACM TOPS*, 25, 2, (May 2022), 1–35.

[15] Lennart Braun, Moritz Huppert, Nora Khayata, Thomas Schneider, and Oleksandr Tkachenko. 2023. Fuse – flexible file format and intermediate representation for secure multi-party computation. Cryptology ePrint Archive, Paper 2023/563. (2023).

[16] Niklas Büscher. 2018. *Compilation for More Practical Secure Multi-Party Computation.* Ph.D. Dissertation. Darmstadt University of Technology, Germany.

[17] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. 2018. HyCC: compilation of hybrid protocols for practical secure computation. In *ACM CCS 2018.* David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, (Eds.) ACM Press, (Oct. 2018), 847–861.

[18] Niklas Büscher and Stefan Katzenbeisser. 2015. Faster secure computation through automatic parallelization. In *24th USENIX Security Symposium (USENIX Security 15).* USENIX Association, (Aug. 2015), 531–546.

[19] David Chaum, Claude Crépeau, and Ivan Damgård. 1988. Multiparty unconditionally secure protocols (extended abstract). In *20th ACM STOC.* ACM Press, (May 1988), 11–19.

[20] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Wenting Zheng. 2023. Silph: a framework for scalable and accurate generation of hybrid mpc protocols. Cryptology ePrint Archive, Paper 2023/060. https://eprint.iacr.org/2023/060. (2023). DOI: 10.1109/SP46215.2023.00103.

[21] KU Leuven COSIC. 2019. SCALE-MAMBA. (2019). https://github.com/KULeuven-COSIC/SCALE-MAMBA.

[22] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13, 4, 451?–490.

[23] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. 2013. Practical covertly secure MPC for dishonest majority - or: breaking the SPDZ limits. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings* (Lecture Notes in Computer Science). Jason Crampton, Sushil Jajodia, and Keith Mayes, (Eds.) Vol. 8134. Springer, 1–18. DOI: 10.1007/978-3-642-40203-6_1.

[24] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings* (Lecture Notes in Computer Science). Reihaneh Safavi-Naini and Ran Canetti, (Eds.) Vol. 7417. Springer, 643–662. DOI: 10.1007/978-3-642-32009-5_38.

[25] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015.* The Internet Society, (Feb. 2015).

[26] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: an efficient SMT solver. In *TACAS 2008.* C. R. Ramakrishnan and Jakob Rehof, (Eds.) Vol. 4963. Springer, (Apr. 2008), 337–340.

[27] Dmitry Duplyakin et al. 2019. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC).* (July 2019), 1–14.

[28] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO 2020, Part II* (LNCS). Daniele Micciancio and Thomas Ristenpart, (Eds.) Vol. 12171. Springer, Heidelberg, (Aug. 2020), 823–852.

[29] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. 2022. CostCO: an automatic cost modeling framework for secure multi-party computation. Cryptology ePrint Archive, Report 2022/332. https://eprint.iacr.org/2022/332. (2022).

[30] Azadeh Farzan and Victor Nicolet. 2021. Phased synthesis of divide and conquer programs. In *ACM PLDI 2021.* Stephen N. Freund and Eran Yahav, (Eds.) ACM, (July 2021), 974–986.

[31] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to play any mental game or A completeness theorem for protocols with honest majority. In *19th ACM STOC.* Alfred Aho, (Ed.) ACM Press, (May 1987), 218–229.

[32] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. SoK: general purpose compilers for secure multi-party computation. In *2019 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, (May 2019), 1220–1237.

[33] Tim Heldmann, Thomas Schneider, Oleksandr Tkachenko, Christian Weinert, and Hossein Yalame. 2021. Llvm-based circuit compilation for practical secure computation. In *Applied Cryptography and Network Security.* Kazue Sako and Nils Ole Tippenhauer, (Eds.) Springer International Publishing, Cham, 99–121.

[34] Muhammad Ishaq, Ana L. Milanova, and Vassilis Zikas. 2019. Efficient MPC via program analysis: A framework for efficient optimal mixing. In *ACM CCS 2019.* Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, (Eds.) ACM Press, (Nov. 2019), 1539–1556.

[35] Ralf Karrenberg. 2015. *Automatic SIMD Vectorization of SSA-based Control Flow Graphs.* Springer. ISBN: 978-3-658-10112-1.

[36] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *ACM CCS 2020.* Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, (Eds.) ACM Press, (Nov. 2020), 1575–1590.

[37] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS 2016.* Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, (Eds.) ACM Press, (Oct. 2016), 830–842.

[38] Marcel Keller, Peter Scholl, and Nigel P. Smart. 2013. An architecture for practical actively secure MPC with dishonest majority. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013.* Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, (Eds.) ACM, 549–560. DOI: 10.1145/2508859.2516744.

[39] Benjamin Levy, Muhammad Ishaq, Ben Sherman, Lindsey Kennard, Ana Milanova, and Vassilis Zikas. 2023. Combine: compilation and backend-independent vectorization for multi-party computation. Cryptology ePrint Archive, Paper 2023/089. https://eprint.iacr.org/2023/089. (2023). https://eprint.iacr.org/2023/089.

[40] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, (May 2015), 359–376.

[41] Payman Mohassel and Peter Rindal. 2018. ABY^3: A mixed protocol framework for machine learning. In *ACM CCS 2018.* David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, (Eds.) ACM Press, (Oct. 2018), 35–52.

[42] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, (May 2017), 19–38.

[43] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. 2016. Frigate: a validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, 112–127.

[44] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. Graphsc: parallel secure computation made easy. In *2015 IEEE Symposium on Security and Privacy*, 377–394.

[45] Tobias Nipkow and Gerwin Klein. 2014. *Concrete Semantics: With Isabelle/HOL.* Springer, Heidelberg, Germany. ISBN: 3319105418.

[46] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. 2020. Unifying compilers for SNARKs, SMT, and more. Cryptology ePrint Archive, Report 2020/1586. https://eprint.iacr.org/2020/1586. (2020).

[47] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: improved mixed-protocol secure two-party computation. In *USENIX Security 2021.* Michael Bailey and Rachel Greenstadt, (Eds.) USENIX Association, (Aug. 2021), 2165–2182.

[48] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, (May 2014), 655–670.

[49] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, 655–670.

[50] Aseem Rastogi, Nikhil Swamy, and Michael Hicks. 2019. Wys*: A DSL for Verified Secure Multi-party Computations. In *Principles of Security and Trust.* Flemming Nielson and David Sands, (Eds.) Springer International Publishing, Cham, 99–122. ISBN: 978-3-030-17138-4.

[51] Michael L. Scott. 2009. *Programming Language Pragmatics (3. ed.)* Academic Press.

[52] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. 2015. TinyGarble: highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy.* IEEE Computer Society Press, (May 2015), 411–428.

[53] Vijay V. Vazirani. 2010. *Approximation Algorithms.* Springer, Heidelberg, Germany. ISBN: 3642084699.

[54] Andrew Chi-Chih Yao. 1982. Protocols for secure computations (extended abstract). In *23rd FOCS.* IEEE Computer Society Press, (Nov. 1982), 160–164.

[55] Samee Zahur and David Evans. 2015. Obliv-c: a language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153. (2015).

[56] Yihua Zhang, Aaron Steele, and Marina Blanton. 2013. PICCO: a general-purpose compiler for private distributed computation. In *ACM CCS 2013.* Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, (Eds.) ACM Press, (Nov. 2013), 813–826.