How To Scale Multi-Party Computation

MARCEL KELLER, CSIRO's Data61, Australia

We propose a solution for optimized scaling of multi-party computation using the MP-SPDZ framework (CCS'20). It does not use manual optimization but extends the compiler and the virtual machine of the framework, thus providing an improvement for any user. We found that our solution improves timings four-fold for a simple example in MP-SPDZ, and it improves an order of magnitude on every framework using secret sharing considered by Hastings et al. (S&P'19) either in terms of time or RAM usage. The core of our approach is finding a balance between communication round optimization and memory usage.

CCS Concepts: • **Security and privacy** → **Cryptography**; **Privacy-preserving protocols**.

Additional Key Words and Phrases: multi-party computation, implementation, compilation

1 INTRODUCTION

Multi-party computation (MPC) is a cryptographic technique that allows a set of parties to compute on their private data without revealing any of it. Results will only be released to some or all parties if they agree to do so. One could think of the computation happening in a black box that is coordinated by the parties. Another mental model is the replacement of a trusted third party by a protocol between the parties. Since its original proposition in the 1980s [3, 8, 16], MPC has emerged as a major privacy-enhancing technology with endorsements by several international organizations [2, 34, 35, 38].

Over the last two decades, a long line of works has focused on improving the practicability of MPC, both in terms of implementations and simpler protocols. The former was started by Malkhi et al. [31] while two examples of the latter are Damgård et al. [11] and Araki et al. [1]. Since Malkhi et al., many frameworks for MPC were released as open-source software. They cover a range of protocols and approaches as covered by Hastings et al. [19].

One reason for this diversity is that some MPC protocols requires an increasing number of communication rounds as the computation grows whereas others only need a constant number of rounds in theory. Yao's garbled circuits [28] are the classical example for the latter. In this work, we will focus more on the former because applications such as privacy-preserving machine learning often use non-constant-round protocols [26].

However, these protocols pose the following challenge: Communication rounds are so expensive that is beneficial to reduce them by combining parallel computation, but it can be prohibitively expensive in terms of memory to reduce the computation to the minimal number of rounds possible. It is therefore imperative to find a trade-off between these two goals, and we propose a new solution to do so based on the commonly used MP-SPDZ framework [21].

At the heart of our solution lies the question of how to represent the computation most efficiently. Common representations of computation are binaries for CPUs or non-machine specific representations such as the internal representation of LLVM [27] or Java [17]. However, these are geared towards the properties of CPUs, which are not encumbered by the issue of communication rounds. MP-SPDZ uses its own bytecode, which allows for an unlimited parallelization of operations reliant on networking. We have extended MP-SPDZ's representation, compiler, and virtual machine to implement our solution, and we show that it makes computation of a simple example more efficient.

Author's address: Marcel Keller, CSIRO's Data61, Australia, marcel.keller@data61.csiro.au,mks.keller@gmail.com.

2 PRELIMINARIES

2.1 Multi-Party Computation

Multi-party computation denotes computation on data that is shared between several parties in a black-box manner, that is, no data input or intermediate data is revealed unless the parties chose to do so. A key technique in MPC is secret sharing, which is used for all intermediate values in a computation. The simplest scheme is additive secret sharing where every party knows a random value x_i such that all the values add up to the secret: $x = \sum_i x_i$. This requires a finite domain for choosing the shares uniformly at random, for example a modular integer domain: $\mathbb{Z}_m = \mathbb{Z}/(m\mathbb{Z})$.

Clearly, adding shares corresponds to adding secrets if addition is transitive, which is the case for the domain mentioned above. It is not known how to multiply additive shares in secret without more complex cryptographic machinery such as oblivious transfer [15, 24] or homomorphic encryption [9]. However, this can be achieved by assigning several additive shares to every party, resulting in a scheme called replicated secret sharing. The simplest case of this features three parties with every parting holding two shares. For a secret x, the i-th party holds $(x_{i+1 \bmod 3}, x_{i-1 \bmod 3})$ such that $x = \sum_{i=0}^{2} x_i$ for random x_i. In addition, if y has been shared in the same way, it holds that

$$\begin{aligned} x \cdot y &= (x_0 + x_1 + x_2) \cdot (y_0 + y_1 + y_2) \\ &= (x_0 \cdot (y_0 + y_1) + x_1 \cdot y_0) \\ &\quad + (x_1 \cdot (y_1 + y_2) + x_2 \cdot y_1) \\ &\quad + (x_2 \cdot (y_2 + y_0) + x_0 \cdot y_2). \end{aligned}$$

Every summand of the last term can be computed by a single party. For example, the last line only contains indices 0 and 2 and can thus be computed by party 1. It follows that the parties can compute an additive secret sharing of the product of two numbers available in replicated secret sharing. To make the result available for further multiplications, the parties have to communicate to turn the additive secret sharing into a replicate one again. Furthermore, they have to randomize the shares because the additive secret is not uniformly random. See Araki et al. [1] for a full specification.

Many operations that are atomic on CPUs, such as comparisons or divisions, have to be implemented using multiplications in MPC. See Catrina and de Hoogh [6, 7] for examples. This raises the question on how to schedule larger computations efficiently in MPC, in particular with regard to networking rounds. For example, it might make sense to run two independent multiplications in parallel to save on network rounds when using the above multiplication (or any other multiplication protocol because it is known that multiplication requires communication). In the following section, we will discuss the various approaches in prior work.

2.2 Approaches for Implementing MPC

2.2.1 The Plain Approach. The straight-forward approach is to simply instruct the MPC framework on the desired operations, which will be executed immediately, including networking operations. Given the cost of network rounds, this imposes considerable onus on the developer for every added computation as they have to put great effort into finding operations that can be parallelized. This approach is used by many implementations of MPC [12, 13, 32, 39].

2.2.2 Online Scheduling. VIFF [10] was the first framework to deviate from the above approach by introducing dynamic scheduling. The core idea there is to maintain a dependency graph of values "behind the scenes", that is, every share or output value seen by the developer in the computation is only a promise of a value available in the future. Whenever an operation on these values is called, a new future value is created, and it is attached to the actual operation to be executed when the actual value arrives via the network.

The cost of this approach is the maintenance of the graph of value dependencies, although the use of Python makes this easy. A further cost comes from the fact that the parties are not synchronized. In a three-party protocol for example, one party might receive the necessary values for two parallel multiplications in a different order than another party. VIFF maintains a program counter to identify values according to the order of their creation. However, this is not straightforward because values are sometimes created only as a reaction to incoming information.

VIFF's successor MPyC [36] uses the same approach and is also based on Python. We are not aware of any framework using a similar approach with a more low-level implementation (e.g., C++). It is therefore hard to make a fair comparison to other approaches they have been implemented in more efficient settings.

2.2.3 Compilation. Some MPC frameworks use a representation of computation, which can be optimized [4, 20, 25, 29, 33, 40, 41]. Keller et al. [25] focus on reducing the number of communication rounds. This approach was later refined in MP-SPDZ [21].

Some of these frameworks involve a full unrolling of all loops, i.e., a loop over n leads to a representation of size $O(n)$. This clearly is not suitable to scaling. For example, CBMC-GC [20] cannot handle a relatively simple example with 1000 data points as shown in Figure 6.

Other frameworks do not seem to undertake an attempt to optimize the network rounds [4, 29, 41]. If the computation is done using garbled circuits as in ObliVM [29], this matters less, however.

Somewhat interestingly, Frigate [33] seems attempting to achieve a trade-off as evidenced by the fact that representation first grows quickly than falls down and grows more slowly with a growing data size. This is reflected in the running times in Figure 6.

In the following section, we will discuss the MP-SPDZ compilation and representation in more detail.

2.3 Compilation in MP-SPDZ

MP-SPDZ stands out in the way that it defines an entire instruction set geared towards many kinds of MPC, totaling more than 200 instructions [22]. The most comparable framework, Sharemind, features about 150 implemented instructions [37]. However, there are notable differences between the two. First, Sharemind instructions are type-independent unlike MP-SPDZ instructions that use different codes for different type combinations. Second, Sharemind uses more instructions for cases where the use could be reduced. For example, there are 11 "jump" instructions including six for all possible comparisons. Lastly, MP-SPDZ has more instructions that are more specific to MPC such as for secure shuffling, matrix multiplication, probabilistic truncation, and edaBit [14] generation.

We will use the example of a simple loop to illustrate MP-SPDZ's compilation process and instruction set. The high-level code in Figure 1 uses a function decorator to avoid unrolling the loop at compile time.

The code features the following instructions:

ldint Load a compile-time integer to a cleartext integer register.

addint Add two integer registers.

ldmsi Load a secret integer register from run-time memory address (cleartext integer register).

stmsi Store a secret integer register to a run-time memory address (cleartext integer register).

muls Multiply secret integers. The additional parameters assist in parallelization.

ltc Compares cleartext integer registers.

```
a, b, c = [sint.Array(10) for i in range(3)]
@for_range(10)
def _(i):
    c[i] = a[i] * b[i]

ldint ci0, 0 # 0
ldint ci3, 8192 # 1
addint ci2, ci0, ci3 # 2
ldmsi s1, ci2 # 3
ldint ci1, 8202 # 4
addint ci3, ci0, ci1 # 5
ldmsi s2, ci3 # 6
muls 4, 1, s0, s1, s2 # 7
ldint ci2, 8212 # 8
addint ci1, ci0, ci2 # 9
stmsi s0, ci1 # 10
ldint ci3, 1 # 11
addint ci2, ci0, ci3 # 12
ldint ci1, 0 # 13
addint ci0, ci2, ci1 # 14
ldint ci3, 10 # 15
ltc ci1, ci2, ci3 # 16
jmpnz ci1, -17 # 17
```

Fig. 1. High- and low-level code for run-time loop in MP-SPDZ. Run-time range checks have been deactivated for simplicity.

```
a, b, c = [sint.Array(10) for i in range(3)]
for i in range(10):
    c[i] = a[i] * b[i]

ldms s10, 8192 # 0
(...)
ldms s29, 8211 # 19
muls 40, 1, s9, s10, s11, 1, s8, s12, s13, (...) # 20
stms s9, 8212 # 21
(...)
stms s0, 8221 # 30
```

Fig. 2. High- and low-level code for compile-time loop in MP-SPDZ. Run-time range checks have been deactivated for simplicity.

jmpnz Jump in the bytecode if the given cleartext integer registers is non-zero.

Lines 1–10 are concerned with the multiplication and the array accesses, while lines 11–17 are concerned with the loop. `ci2` defined on line 12 grows one bigger with every execution of the loop body, and after 10 loop bodies, the comparison on line 16 results 0, avoid jumping back on line 17 for the first time. In contrast, Figure 2 shows the equivalent code with Python loop that is unrolled at compile time.

Lastly, Figure 3 shows the same functionality using MP-SPDZ's vector capabilities. This is clearly the most compact code, but it puts the onus on the programmer by using unusual structures. While it is round-optimal as it allows the

```
a, b, c = [sint.Array(10) for i in range(3)]
c[:] = a[:] * b[:]

vldms 10, s10, 8192 # 0
vldms 10, s20, 8202 # 1
muls 4, 10, s0, s10, s20 # 2
vstms 10, s0, 8212 # 3
```

Fig. 3. High- and low-level code for vectorized computation in MP-SPDZ. Run-time range checks have been deactivated for simplicity.

virtual machine to execute all operations in parallel, it might use too much memory when scaling up because it requires all intermediate protocol information to be held at once. In Section 4, we will explore a way of scaling up while striking a balance between memory usage, network rounds, and programmer involvement.

3 AN EXAMPLE STUDY

We will use the cross-tabulation example by Hastings et al. [19]. The task can be understood as summing the salaries of employees by same category (e.g., age or location). The input is given as two tables, one linking unique identifiers to the salary, and another linking the same identifiers to a category identifier. A simple solution is given in Algorithm 1. While it is not the most efficient algorithm for plaintext computation, it is possible to implement it in secure computation without the use of sophisticated data structures that are more challenging in secure computation. Furthermore, the purpose of this work is assessing the scalability rather finding the most efficient approach for the example.

Algorithm 1: Cross-tabulation

```
    Input  Salary table, category table
    Output  Sum of salaries for every category
1: Initialize output table
2: for Every ID in the salary table do
3:     for Every ID in the category table do
4:         if IDs are the same then
5:             Increase salary sum of the category in the output table
6:         end if
7:     end for
8: end for
```

Consider the scheduling in the context of multi-party computation. The comparisons on line 4 are independent of anything else; they thus could be computed in parallel at the beginning. However, this might exhaust the storage as it would require quadratic storage for all the results. On the other hand, running the comparison strictly one after leads to a large number of communication rounds in non-constant round protocols while likely not underutilizing the memory. The optimal solution is therefore most likely in a solution that computes some comparisons in parallel but not all of them. To this end, Büscher et al. [5] have proposed a measured loop unrolling that unrolls the loop partially until a specified budget is reached. The partially unrolled loop is then optimized for communication rounds, that is, all the comparisons in the partially unrolled loop are scheduled in parallel in our example. MP-SPDZ [21] implements this approach by using the number of instructions in its internal representation as the relevant parameter for the budget.

4 SCALING COMPUTATION IN MP-SPDZ

MP-SPDZ executes high-level code in Python at compilation time. This means that Python loops are fully executed, or in other words, completely unrolled. Clearly, this does not scale well as the size of the representation grows with the size of the dataset. The other extreme is to not unroll the loop at all. While this keeps the representation of constant size, not only the number of communication rounds goes up but also the average communication per round remains at a very low level. This means that the wall clock time is dominated by the number of rounds (which could be lowered) as opposed to the amount of communication (which is determined by the protocol and the computation). Algorithms 2 and 3 detail the two procedures. They are illustrated by the two code examples in Section 2.3.

Algorithm 2: Compilation with unrolling

```
Input  Loop in MP-SPDZ high-level code
Output  MP-SPDZ bytecode without jump instructions
1: for every loop iteration do
2:     Translate the high-level code to the internal representation
3: end for
4: Optimize the entire internal representation at once
5: return bytecode of internal representation
```

Algorithm 3: Compilation without unrolling

```
Input  Loop in MP-SPDZ high-level code
Output  MP-SPDZ bytecode with jump instructions
1: Translate the high-level code of one loop body to the internal representation
2: Optimize the loop body alone
3: Add jump instructions to repeat execution of loop body
4: return bytecode of internal representation
```

Figures 4 and 5 show the running time and the bytecode size for the two approaches. As expected, loop unrolling decreases the time but increases the bytecode size. We have used the three-party computation protocol suite in Appendix A of Keller and Sun [26] for the timing, and we have run the three party on a single machine. While this does not reflect the cost in a more realistic setting, it does reflect the computational cost. We have used a single thread per party.

Figure 5 also demonstrates that, once the bytecode size would near 1 GB, the compiler cannot handle it memory anymore, hence the lack of figures for the unrolling approach above a dataset size of 128.

The fact that unrolling leads to almost one order of magnitude savings in time raises the desire to achieve a compromise between the two approaches. HyCC [5] partially provides this compromise by proposing a budgeting

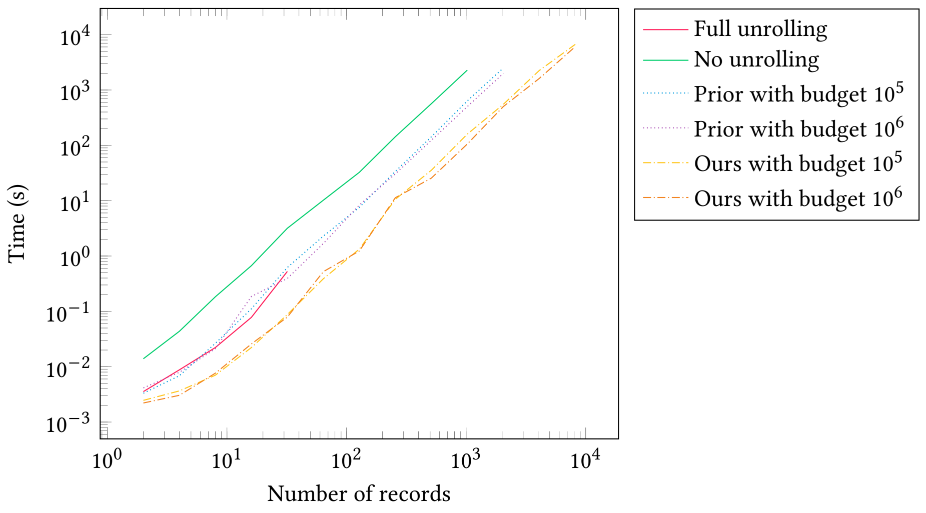

Fig. 4. Time for running different variants with cross-tabulation in 3PC on one machine.

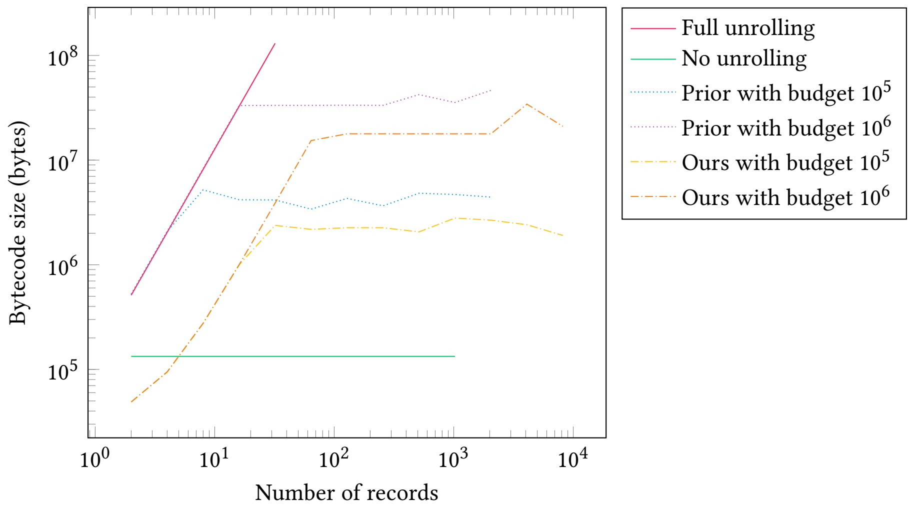

Fig. 5. Bytecode size for different variants with cross-tabulation in 3PC.

approach: Loops are unrolled until some threshold is crossed. The partially unrolled loop is then optimized for the number of rounds before completing the unrolling without optimization. The paper is not entirely clear about the last step, but we conclude this from the code because the underlying circuit representation does not support jumps which would be required for run-time loops.

In contrast, MP-SPDZ supports jump instructions which make it possible to have loops that are not completely unrolled at compile time. Algorithm 4 explains the procedure for creating run-time loops that are still optimized to some extent, and Algorithm 5 details its extension to nested loops.

Figures 4 and 5 also show results for the MP-SPDZ implementation thereof with budget 10^5 and 10^6. The number refers to the number of instruction calls in the unrolled loop. The approach proves to be as competitive on time as unrolling while maintaining an upper limit on the bytecode.

Algorithm 4: Compilation with budget

```
    Input  Loop in MP-SPDZ high-level code
    Output  MP-SPDZ bytecode potentially with jump instructions
1: while size of internal representation is within the budget do
2:     Translate the high-level code to the internal representation
3: end while
4: Optimize the internal representation created so far at once
5: Add jump instructions to complete the loop execution if necessary
6: return bytecode of internal representation
```

Algorithm 5: Compilation with budget across nested loops

```
    Input  Nested loops in MP-SPDZ high-level code
    Output  MP-SPDZ bytecode with jump instructions
1: while size of internal representation is within the budget do
2:     while there is an unprocessed loop do
3:         Translate the innermost loop body to the internal representation
4:     end while
5: end while
6: Optimize the internal representation at once
7: Add jump instructions to complete the loop execution for remaining loops if any
8: return bytecode of internal representation
```

However, it is not entirely satisfying as it involves transcribing the same operation (equality testing) repeated to the low-level operations supported by the MP-SPDZ virtual machine design. This is where our contribution comes in. We added a call facility to the virtual machine, and used this call facility to call the same code for equality testing whenever needed. This considerably improves the timing and slightly decreases the bytecode size compared to the prior approach.

An added benefit is that only this approach allows for vectorization without the programmer having to take care of it. This means that a programmer can specify single-value comparisons as on line 4, and the compiler first treats them atomically (i.e., without turning them into low-level instructions). Only after seeing how many equality tests to execute in parallel, the compiler creates a vectorized function for equality testing and adds a call to this function. This way, the compiler never considers more than one execution of equality testing, which reduces the number of instructions in the internal representation and thus the memory usage. Algorithm 6 outlines our approach as pseudocode.

Algorithm 6: Compilation with budget and placeholders

```
Input  Loop in MP-SPDZ high-level code, list of operations to retain
Output  MP-SPDZ bytecode with jump and call instructions
1: while size of internal representation is within the budget do
2:     Translate the high-level code to the internal representation but retain operations on input list as a single instruction call
3: end while
4: Optimize the internal representation created so far at once. This might include vectorization of retained operations.
5: Optimize the retained operations individually and add call instructions accordingly
6: Add jump instructions to complete the loop execution
7: return bytecode of internal representation
```

5 OTHER FRAMEWORKS

Hastings et al. [19] have started a GitHub repository [18] with Docker images to easily run every framework considered in their paper. It has been extended by contributors to include further frameworks. The repository also includes an implementation of the cross-tabulation example used in this work. Figure 6 shows the time it takes a selection of frameworks to compile and execute the cross-tabulation example depending on the dataset size. We have run them up to one hour or until the memory usage exceeded 10 GB. The latter explains why some stop far below one hour. The best-performing framework is EMP [39], which implements Yao's garbled circuits. These require constant rounds of communication, which is why they are less affected by the trade-off elaborated on in this work. This also holds for Obliv-C [40]. All other frameworks perform worse than our solution based on MP-SPDZ, including the emulation of Sharemind.

An usual case is using SecretFlow [30] with JAX, which allows for maximum parallelization, corresponding to full unrolling in Section 4. This approach quickly exhausts the memory as the dataset grows. It also is less intuitive than the other code examples, see Figure 7.

REFERENCES

[1] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *ACM CCS 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM Press, 805–817. https://doi.org/10.1145/2976749.2978331

[2] Bank for International Settlements. 2023. Project Aurora: The power of data, technology and collaboration to combat money laundering across institutions and borders. (2023). https://www.bis.org/publ/othp66.pdf

[3] 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, 1–10. https://doi.org/10.1145/62212.62213

[4] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS 2008 (LNCS, Vol. 5283)*, Sushil Jajodia and Javier López (Eds.). Springer, Berlin, Heidelberg, 192–206. https://doi.org/10.1007/978-3-540-88313-5_13

[5] 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, 847–861. https://doi.org/10.1145/3243734.3243786

[6] Octavian Catrina and Sebastiaan de Hoogh. 2010. Improved Primitives for Secure Multiparty Integer Computation. In *SCN 10 (LNCS, Vol. 6280)*, Juan A. Garay and Roberto De Prisco (Eds.). Springer, Berlin, Heidelberg, 182–199. https://doi.org/10.1007/978-3-642-15317-4_13

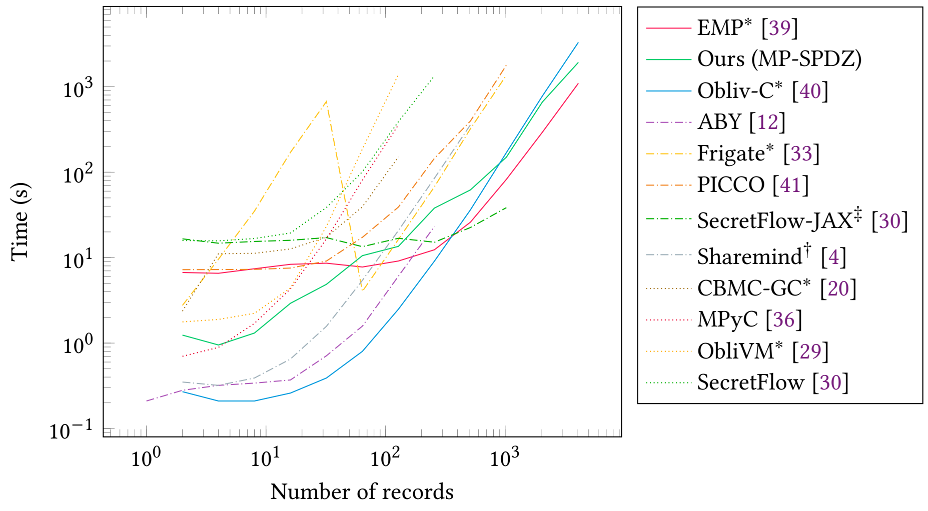

Fig. 6. Time to compile and run cross-tabulation with different MPC frameworks. * denotes a garbled circuit implementation, and † denotes an emulation. ‡ denotes total unrolling, see the text for details. To improve readability, the results are ordered by performance in three groups with the same line style.

```
def map_i(i):
    def map_j(j):
        def map_k(k):
            return (xid[i] == yid[j]) * (cats[j] == k) * vals[i]
        return jax.vmap(map_k)(jnp.arange(num_k))
    return jnp.sum(jax.vmap(map_j)(jnp.arange(n)), axis=0)
res = jnp.sum(jax.vmap(map_i)(jnp.arange(n)), axis=0)
```

Fig. 7. Cross-tabulation with SecretFlow and JAX.

[7] Octavian Catrina and Sebastiaan de Hoogh. 2010. Secure Multiparty Linear Programming Using Fixed-Point Arithmetic. In *ESORICS 2010 (LNCS, Vol. 6345)*, Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou (Eds.). Springer, Berlin, Heidelberg, 134–150. https://doi.org/10.1007/978-3-642-15497-3_9

[8] David Chaum, Claude Crépeau, and Ivan Damgård. 1988. Multiparty Unconditionally Secure Protocols (Extended Abstract). In *20th ACM STOC*. ACM Press, 11–19. https://doi.org/10.1145/62212.62214

[9] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. 2001. Multiparty Computation from Threshold Homomorphic Encryption. In *EUROCRYPT 2001 (LNCS, Vol. 2045)*, Birgit Pfitzmann (Ed.). Springer, Berlin, Heidelberg, 280–299. https://doi.org/10.1007/3-540-44987-6_18

[10] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. 2009. Asynchronous Multiparty Computation: Theory and Implementation. In *PKC 2009 (LNCS, Vol. 5443)*, Stanislaw Jarecki and Gene Tsudik (Eds.). Springer, Berlin, Heidelberg, 160–179. https://doi.org/10.1007/978-3-642-00468-1_10

[11] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO 2012 (LNCS, Vol. 7417)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer, Berlin, Heidelberg, 643–662. https://doi.org/10.1007/978-3-642-32009-5_38

[12] 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. https://doi.org/10.14722/ndss.2015.23113

[13] JIFF development team. 2018. JavaScript library for building web-based applications that employ secure multi-party computation. https://github.com/multiparty/jiff.

[14] 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, Vol. 12171)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer, Cham, 823–852. https://doi.org/10.1007/978-3-030-56880-1_29

[15] Niv Gilboa. 1999. Two Party RSA Key Generation. In *CRYPTO'99 (LNCS, Vol. 1666)*, Michael J. Wiener (Ed.). Springer, Berlin, Heidelberg, 116–129. https://doi.org/10.1007/3-540-48405-1_8

[16] 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, 218–229. https://doi.org/10.1145/28395.28420

[17] James Gosling. 2000. *The Java language specification.* Addison-Wesley Professional.

[18] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. Sample code and build environments for MPC frameworks. https://github.com/MPC-SoK/frameworks.

[19] 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, 1220–1237. https://doi.org/10.1109/SP.2019.00028

[20] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. 2012. Secure two-party computations in ANSI C. In *ACM CCS 2012*, Ting Yu, George Danezis, and Virgil D. Gligor (Eds.). ACM Press, 772–783. https://doi.org/10.1145/2382196.2382278

[21] 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, 1575–1590. https://doi.org/10.1145/3372297.3417872

[22] Marcel Keller. 2020. MP-SPDZ instruction set. https://mp-spdz.readthedocs.io/en/latest/instructions.html.

[23] Marcel Keller. 2024. Using High-Level Functionality in C++. https://mp-spdz.readthedocs.io/en/latest/function-export.html.

[24] 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, 830–842. https://doi.org/10.1145/2976749.2978357

[25] Marcel Keller, Peter Scholl, and Nigel P. Smart. 2013. An architecture for practical actively secure MPC with dishonest majority. In *ACM CCS 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, 549–560. https://doi.org/10.1145/2508859.2516744

[26] Marcel Keller and Ke Sun. 2022. Secure Quantized Training for Deep Learning. In *Proceedings of the 39th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (Eds.). PMLR, 10912–10938. https://proceedings.mlr.press/v162/keller22a.html

[27] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[28] Yehuda Lindell and Benny Pinkas. 2009. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology* 22, 2 (April 2009), 161–188. https://doi.org/10.1007/s00145-008-9036-8

[29] 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, 359–376. https://doi.org/10.1109/SP.2015.29

[30] Junming Ma, Yancheng Zheng, Jun Feng, Derun Zhao, Haoqi Wu, Wenjing Fang, Jin Tan, Chaofan Yu, Benyu Zhang, and Lei Wang. 2023. SecretFlow-SPU: A Performant and User-Friendly Framework for Privacy-Preserving Machine Learning. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 17–33. https://www.usenix.org/conference/atc23/presentation/ma

[31] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay - Secure Two-Party Computation System. In *USENIX Security 2004*, Matt Blaze (Ed.). USENIX Association, 287–302.

[32] 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, 35–52. https://doi.org/10.1145/3243734.3243760

[33] 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. https://doi.org/10.1109/EuroSP.2016.20

[34] OECD. 2023. Emerging privacy-enhancing technologies. 351 (2023). https://doi.org/10.1787/bf121be4-en

[35] Royal Society. 2023. From privacy to partnership: the role of Privacy Enhancing Technologies in data governance and collaborative analysis. (2023). https://royalsociety.org/topics-policy/projects/privacy-enhancing-technologies/

[36] Barry Schoenmakers. 2019. MPyC: Multiparty Computation in Python. https://github.com/lschoe/mpyc.

[37] Sharemind. 2015. Sharemind documentation. https://github.com/sharemind-sdk/vm_m4/blob/master/doc/bytecode.md.

[38] United Nations. 2023. United Nations Guide on Privacy-Enhancing Technologies for Official Statistics. (2023). https://unstats.un.org/bigdata/task-teams/privacy/guide/

[39] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

[40] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. Cryptology ePrint Archive, Report 2015/1153. https://eprint.iacr.org/2015/1153

[41] 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, 813–826. https://doi.org/10.1145/2508859.2516752

A TECHNICAL REFERENCE

This section links the concepts in this paper to places in the MP-SPDZ source code at the time of writing.

Round optimization. The `Tape` class in `Compiler/program.py` holds all instructions for one bytecode object by basic blocks, which in turn contain a sequence of instructions without jumps. The `optimize` method runs an instance of the `Merger` class in `Compiler/allocator.py` for every in basic block. The `Merger` class figures out dependencies between instructions to see which communication-inducing instructions such as multiplication can be merged. See prior work [21, 25] for details. The class also contains code for eliminating dead code, that is, code whose results are not used in any way. This is not activated by default as to not disrupt benchmarking but can be activated using the `--dead-code-elimination` with the compiler.

Loop unrolling with budget. While the documented function for this is `for_range_opt` in `Compiler/library.py` (to be used as function decorator), most of the logic sits in `map_reduce_single` in the same file, in particular the "else" portion of the condition on `n_parallel is not None`. The code not only involves the unrolling of the loop but also stitching basic blocks together in order to allow for further round optimization. Another issue solved are memory accesses conditioned on the loop counter that are very common in code. The loop counter is a run-time variable (`regint`), which makes it impossible for the compiler link memory accesses when optimizing rounds. The `RegintOptimizer` class in `Compiler/allocator.py` makes sure that different `regint` variables containing the same number (as seen by static analysis) are merged into the same `regint` or compile-time constant. This in turn makes sure that the order of memory accesses on the same address is preserved during the round optimization.

Run-time functions. `Compiler/library.py` contains several subclasses of `Function` for different purposes. `FunctionTape` is the oldest one and represents a function to be called in a different thread while `FunctionCallTape` represents a function to be called in the same thread. The latter makes use of the virtual machine capabilities introduced in version 0.3.9. Based on it, `ExportFunction` represents a function to be called from C++ code [23]. Lastly, `FunctionBlock` represents an earlier way of calling functions within the same thread without the capabilities of version 0.3.9. We discourage its use due issues with register allocation. The classes are associated with functions to be used as decorators to turn Python functions into run-time functions. In addition, `method_call_tape` allows turning class method into run-time functions executed in the same thread.

Retained operations. `MergeCISC` in `Compiler/instructions_base.py` represents retained by emulating an instruction that can be merged. This instruction is then turned into instructions that the virtual machine actually provides.[1] The default tool to do so is the the in-thread run-time function using the post-0.3.9 capabilities, with the pre-0.3.9 capabilities as fallback. Functions can be marked as retained operations using the following decorators:

`cisc` for functions that take the outputs as uninitialized registers. There is an optional parameter for the number of outputs (default is 1). This is used for older functionality such as `LTZ` in `Compiler/comparison.py`.

`ret_cisc` for functions where the output is a returned register.

`sfix_cisc` for functions where the first argument and the return value are of type `sfix`. This is used for the mathematical functions in `Compiler/mpc_math.py`.

`bit_cisc` for bit decomposition where the number of bits is an argument.

[1]The virtual machine for emulation provides some of these instructions directly for more efficient emulation.