Symphony: A Concise Language Model for MPC

Presented as a short paper at Foundations of Computer Security (FCS), June 2021.

Ian Sweet
University of Maryland
ins@cs.umd.edu

David Darais
Galois, Inc.
darais@galois.com

David Heath
Georgia Institute of Technology
heath.davidanthony@gmail.com

Ryan Estes
University of Vermont
restes@uvm.edu

William Harris
Galois, Inc.
wharris@galois.com

Michael Hicks
University of Maryland
mwh@cs.umd.edu

Secure Multiparty Computation (MPC) is a subfield of cryptography that allows mutually untrusting parties to compute arbitrary functions over their private inputs while revealing nothing except the function output. While the idea for MPC is old, it is rapidly advancing, with high performance cryptographic back ends, and front-ends based on familiar programming models (e.g., imperative, functional), and languages (Java, C, and C++) [1, 3, 5, 7, 8, 10].

Unfortunately, while these languages allow the programmer to effectively express many MPCs, they fall short when confronted with problems that require non-trivial *coordination*. Overwhelmingly, MPC languages take the default view that all parties perform the same synchronized activity, SIMD-style. But this approach has problems when trying to scale up to many parties, and when the computation may proceed based on dynamic outcomes.

For example, suppose we wish to implement "March Madness Millionaires" (MMM) where $N = 2^K$ parties take part in a pairwise competition to see who is the richest, with the winners of one round pairing up in the next round until there is a final winner; all $N - 1$ contest winners are made public . Unfortunately, 2-party frameworks like Obliv-C [10] and EMP [8], and N-party languages, such as PICCO [11], Sharemind [2], Frigate [5], and SCALE-MAMBA [1], provide no help in coordinating this sort of task; they take the default view that all parties perform the same synchronized activity. For MMM, parties are pairwise (not N-way) synchronized to start, and require coordination after each round. These languages would allow expressing the "am I richer than you" computation between two parties, but the coordination logic amongst them must be coded up separately, in a separate language. In addition to increasing the chances of a mistake, using a separate language for coordination increases the chances that a mistake is security relevant. Ideally, all of a secure computation should be examined for correctness, and it is easier to do this if the whole computation is in one place.

To support coordination as part of secure computation, we have been developing Symphony, a domain-specific language that supports MPCs with complex coordination. Symphony provides first class support for coordinating parties. In particular, its first class *shares* allow the programmer to elegantly interleave encrypted computation with cleartext local computation, and its `par` expressions allow the programmer to easily control which parties are in scope. SIMD-style sub-computations may take place over *bundles*, which allow different parties to have their own private values. Crucially, Symphony supports a *single-threaded interpretation* of distributed programs: The developer can understand her program as if it runs on a single thread of execution on a single machine, and Symphony guarantees that this is a faithful understanding of the program, even when the program is executed in a distributed setting. This interpretation allows the programmer to *reason effectively* and to avoid bugs while coordinating parties in arbitrary ways.

Symphony was inspired by earlier work on Wysteria [6], which also tackled the coordination problem, and provided a single-threaded interpretation. For example, Wysteria also has `par` blocks, and bundles for SIMD computations. However, Symphony is simpler while still being more powerful. Wysteria places secure computations in special `sec` expressions which have complicated rules about their use. A `sec` expression is translated to a circuit that is run by an MPC backend (e.g., Yao [9] or GMW [4]) with its final result revealed. By contrast, Symphony's shares are essentially suspended circuits, built up iteratively by the program; only when passed to a reveal construct is the circuit executed by the backend. We found that computations like recursive GCD simply could not be expressed in Wysteria.

We have formalized Symphony in a core calculus, which we call λ-Symphony. Using it, we proved that the single-threaded semantics faithfully represents the actual distributed semantics. We note that Symphony's simplicity, compared to Wysteria, is also evident in its formal model, which is far simpler than Wysteria's.

We have also implemented an interpreter for Symphony and used it to implement a variety of MPC programs. Our interpreter is written in Haskell, and connects to the EMP toolkit for the MPC backend (implementing Yao's garbled circuits). We have implemented the same programs in Obliv-C, a state of the art MPC framework for two party computation, which uses C as a front end. For a robust comparison, we have also modified Obliv-C to connect to the EMP toolkit as an alternative MPC backend. We have measured the gate counts and end-to-end execution time of both Symphony and Obliv-C on five different benchmarks: hamming distance, bio-matching, db-analytics, gcd, and edit-distance. Our preliminary results show that both gate counts and end-to-end execution time are similar between Obliv-C and Symphony. (And of course you can't write MMM in Obliv-C.)

Symphony is active work. We are working on

- Developing a static type system for Symphony. At present, Symphony's interpreter is dynamically checked. This means that some failures will not be detected until run-time. A sufficiently powerful type system, for the programs we have implemented, will require both polymorphically constrained and dependent types, and poses an interesting challenge.

- Proving that λ-Symphony enjoys a variant of noninterference — the only information releases occur where the programmer has specified them. We aim to prove this both for the dynamically typed system and the statically typed one.
- Extending the support to additional backends, in particular N-party GMW, to complement 2-party Yao's.
- Implementing more programs, especially those that benefit from added coordination support. Some examples include gaussian elimination, K-means, and private set intersection, as well as multi-round protocols, e.g., for auctions.

REFERENCES

[1] Abdelrahaman Aly, Marcel Keller, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. 2019. SCALE-MAMBA. https://homes.esat.kuleuven.be/ nsmart/SCALE/.

[2] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS 2008: 13th European Symposium on Research in Computer Security (Lecture Notes in Computer Science)*, Sushil Jajodia and Javier López (Eds.), Vol. 5283. Springer, Heidelberg, Germany, Málaga, Spain, 192–206. https://doi.org/10.1007/978-3-540-88313-5_13

[3] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. 2014. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. In *Compiler Construction (Lecture Notes in Computer Science)*, Albert Cohen (Ed.), Vol. 8409. Springer Berlin Heidelberg, 244–249. https://doi.org/10.1007/978-3-642-54807-9_15

[4] 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 Annual ACM Symposium on Theory of Computing*, Alfred Aho (Ed.). ACM Press, New York City, NY, USA, 218–229. https://doi.org/10.1145/28395.28420

[5] 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)*. IEEE, 112–127.

[6] 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, Berkeley, CA, USA, 655–670. https://doi.org/10.1109/SP.2014.48

[7] Berry Schoenmakers. 2019. MPyC: Secure multiparty computation in Python. Github. https://github.com/lschoe/mpyc

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

[9] Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *23rd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Chicago, Illinois, 160–164. https://doi.org/10.1109/SFCS.1982.38

[10] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. Cryptology ePrint Archive, Report 2018/706. https://eprint.iacr.org/2015/1153.

[11] Yihua Zhang, Aaron Steele, and Marina Blanton. 2013. PICCO: a general-purpose compiler for private distributed computation. In *ACM CCS 2013: 20th Conference on Computer and Communications Security*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, Berlin, Germany, 813–826. https://doi.org/10.1145/2508859.2516752