A Generic Hybrid 2PC Framework with Application to Private Inference of Unmodified Neural Networks
(Extended Abstract)

Lennart Braun
Aarhus University
braun@cs.au.dk

Rosario Cammarota
Intel Labs
rosario.cammarota@intel.com

Thomas Schneider
TU Darmstadt
schneider@encrypto.cs.tu-darmstadt.de

Abstract

We present a new framework for generic mixed-protocol secure two-party computation (2PC) and private evaluation of neural networks based on the recent MOTION framework (Braun et al., ePrint '20). We implement five different 2PC protocols in the semi-honest setting – Yao's garbled circuits, arithmetic and Boolean variants of Goldreich-Micali-Wigderson (GMW), and two secret-sharing-based protocols from ABY2.0 (Patra et al., USENIX Security '21) – together with 20 conversions among each other and new optimizations. We explore the feasibility of evaluating neural networks with 2PC without making modifications to their structure, and provide secure tensor data types and specialized building blocks for common tensor operations. By supporting the Open Neural Network Exchange (ONNX) file format, this yields an easy-to-use solution for privately evaluating neural networks, and is interoperable with industry-standard deep learning frameworks such as TensorFlow and PyTorch. By exploiting the networks' high-level structure and using common 2PC techniques, we obtain a performance that is comparable to that of recent, highly optimized works and significantly better than when using generic 2PC for low-level hybrid circuits.

1 Introduction

Secure Computation Secure multi-party computation (MPC) allows a set of mutually distrusting parties to evaluate a functionality on their private inputs and obtain private outputs such that each party learns only what it can deduce from its own input and output, [21]. The two-party case is also designated as secure two-party computation (2PC). MPC allows computation on data which the owners are either reluctant or not allowed to share with each other, e.g., due to data privacy regulations. Use cases include computing statistics on sensitive medical and corporate data [14], or outsourcing of computation to cloud services [25]. The concept of secure computation dates back to the 1980s [47, 46, 18], and has become increasingly practical over the last two decades [34, 39].

Machine Learning As a completely different technology, machine learning techniques such as deep neural networks have been applied to many different tasks over the last years, e.g., image classification [19]. Since training neural networks requires large datasets and significant computational

Submitted to Privacy in Machine Learning (PriML) 2021. Do not distribute.

resources, companies might be inclined to keep the resulting models secret to protect their investment. On the other hand, customers may be hesitant to just pass their input to the service provider.

Privacy-Preserving Machine Learning The combination of multi-party computation and machine learning allows a variety of privacy-preserving online services [32, 23, 4, 8]. For example, to classify a picture, a customer can run an MPC protocol with a service provider instead of just uploading the picture. It then gets the result of the classification without having to reveal the input, while the provider can keep its model secret.

Our Contributions In this work, we present the first secure two-party computation framework combining *five different protocols* such that shared data can be converted arbitrarily among the protocols. In the framework, we combine Yao's protocol [46, 31], the arithmetic and Boolean variants of GMW [18] and the secret-sharing-based arithmetic and Boolean protocols from ABY2.0 [38]. We analyze the protocols, and introduce new *optimizations* and conversion protocols. We integrate the protocols into the recent MPC framework MOTION [7] which is partially redesigned and extended. Furthermore, we explore the feasibility of private neural network inference using *standard MPC techniques* without making modification to the networks. To this end, we discuss protocols for common neural network operations, and implement them as specialized building blocks in MOTION. By supporting the *ONNX* file format [37] we enable *interoperability with deep learning frameworks used in industry* such as TensorFlow and PyTorch. Finally, we evaluate the performance of our framework using various benchmarks, and compare the implemented protocols among themselves and with prior work. The specialized building blocks perform clearly better for neural networks than the equivalent generic 2PC protocols for hybrid circuits, and we achieve a performance comparable to recent, highly optimized works such as GAZELLE [24] and DELPHI [35].

Related Work The most relevant works regarding generic MPC are ABY [13], MOTION [7], and ABY2.0 [38] upon which we heavily build (see § 2, § 3). TASTY [22], ABY and MOTION are software frameworks for 2PC and MPC in the semi-honest security model supporting Boolean and arithmetic operations. MP-SPDZ [26] and SCALE-MAMBA [2] provide protocols that are secure in the malicious model. EzPC [10] and HyCC [9] can compile high-level function descriptions into hybrid circuits usable for MPC. There is a large body of work on private evaluation of neural networks, e.g. [4, 16, 36, 24, 42, 41]. Many of these modify the networks, e.g., by quantizing weights, to obtain more efficient protocols, whereas we try to preserve the original network as much as possible.

Organization In § 2, we give an overview of the used protocols. § 3 covers our improvements to the MOTION framework, and in § 4, we discuss the results of our experimental evaluation. Supplementary material can be found in the appendix.

2 Protocols

In this work, we consider *five* generic 2PC protocols for Boolean circuits (denoted with Y, B, β) and arithmetic circuits over rings $\mathbb{Z}_{2^\ell}$ (denoted with A, α). The protocols are secure in the semi-honest security setting, i.e., corrupted parties follow the protocol, but try to learn additional information. As in ABY [13], we use Yao's garbled circuit protocol (Y) [47, 46] and the arithmetic (A) and Boolean (B) variants of the GMW protocol [18]. Additionally, we combine these with the new arithmetic (α) and Boolean (β) protocols introduced in ABY2.0 [38]. See § A.2 for a short summary of the protocols. For all protocol operations, we consider single instruction multiple data (SIMD) variants. They operate element-wise on vectors of values, and can be implemented much more efficiently.

Conversions Some operations (e.g. additions, multiplications) are naturally expressed as arithmetic circuits, and others are more efficiently represented in Boolean circuits (e.g. comparisons). Hence, it is often advantageous to combine both kinds into a hybrid circuit. To evaluate such a hybrid circuit with 2PC protocols, we need to convert between different representations. We provide *conversions between all five protocols*. Many are based on prior work [13, 38], some are new or optimized.

Neural Networks We use the generic 2PC protocols to securely evaluate the *tensor operations* in *neural networks*. Here, we encode the values in $\mathbb{Z}_{2^\ell}$ using a fixed-point representation and the truncation protocol by [36] instead of using floating-point numbers. Since some of the operations are more efficiently computed with an arithmetic protocol and others with a Boolean protocol, we use the conversions to change the data representation as necessary. For fully-connected and convolutional layers, we use generalizations of the integer multiplication in the A and α protocols (cf. [36]). AvgPool is a linear operation and needs only a truncation. MaxPool is based on optimized Boolean circuits. For the ReLU activation function we provide different variants, e.g., based on special bit-integer multiplication protocols. More details about these tensor operations are given in § A.3.

ABY vs. ABY2.0 We compare the ABY [13] (A, B, Y) and ABY2.0 [38] (α, β, Y) protocol suites: Storing an α shares requires twice the space compared to an A share. Linear operations are computed in both cases locally without interaction, although ABY2.0 requires more (local) computation during the setup phase. The ABY2.0 *multiplication* needs only half of the online communication compared to GMW [38]. More generally, the online communication in GMW is linear in the size of the inputs, whereas for ABY2.0, it is linear in the output size. This also holds when the multiplication protocols are generalized, e.g., to matrix multiplications or convolutions.

The *setup phase*, i.e., the part of the protocol that can be executed before the inputs are known, of GMW depends only on the number of multiplications. For ABY2.0, it also depends on the circuit structure, which makes it more costly in general. In the case of neural networks, the (matrix) multiplication operations are relatively few and relatively large compared to integer multiplications in a normal arithmetic circuit. Since MaxPool and ReLU layers repeatedly apply the same basic function, the setup can be computed in batches with SIMD operations. Thus, the disadvantages of a function-dependent setup phase do not carry much weight in the case of neural networks.

We compare the *conversions* among the A, B, and Y protocols to those among α, β, and Y. The latter were presented by [38] who also showed that they compare favorably to the original ABY conversions [13]. The newer conversions presented in our work improve on ABY [13] and the differences have become smaller. Looking at costs, a pattern becomes apparent: The conversions from A/B to Y require two rounds while the other direction is free of communication in the online phase. On the other hand, converting from α/β to Y and vice versa costs one round. Overall, in a deep circuit (e.g., a neural network) where different layers are computed alternately with Y and one of A/α, the number of rounds in the online phase are the same.

We examine different ways to compute *ReLU layers* in neural networks. The online communication costs for the ABY2.0 protocols are significantly lower compared to their GMW counterparts. The total communication costs are either similar or slightly in favor of ABY2.0.

3 Extending MOTION for Neural Networks

Extending the Framework In this work, we build upon and extend the MOTION framework for mixed-protocol multi-party computation [7]. It is not only a library implementing some MPC protocols, but also a *framework* that consists of useful components which can be composed and extended as needed. We implement the five generic 2PC protocols (§ 2) and conversions among them, as well as building blocks for neural network evaluation.

We redesigned many of the framework's interfaces to reduce overhead and *improve flexibility* by decoupling the components (see § B for more details). Based on the terminology of circuits, MOTION [7] implements the primitive operations of protocols in `Gate` classes, which it evaluates using fibers, i.e., threads that run in user space. We extracted the execution code into `Executor` classes that take a collection of gates and execute them according to some strategy. This improves the framework's flexibility by allowing the implementation of different strategies. As in [7], the default executor for general-purpose MPC with arbitrary circuits creates fibers for all gates which are then executed by a thread pool.

MOTION for Neural Networks For evaluating *neural networks*, we take a different approach: Many neural networks have a simple or even straight-line topology and are composed from a small set of different tensor operations (e.g., linear and pooling layers and activation functions). Hence, instead of compiling their descriptions into low-level circuits, we provide optimized building blocks that directly implement common high-level operations on shared tensors. The tensor operations are more computationally complex than the primitive operations of a circuit, but they have a very regular structure, e.g., a *single* activation function is applied to *all* elements of a tensor. We exploit this knowledge and use a specialized executor to evaluate the tensor operations sequentially, while parallelizing the operations themselves with multi-threading and SIMD operations.

We enable *interoperability* with industry standard deep learning frameworks such as TensorFlow and PyTorch by supporting the ONNX file format [37], an open standard for deep learning models. Our new `OnnxAdapter` based on the ONNX library[1] can parse the description of neural networks and automatically constructs the respective tensor operations in MOTION. This makes it easy to privately evaluate models built with common deep learning frameworks. Moreover, we built tools to display information about the neural network and estimate the costs of a secure evaluation.

4 Performance Evaluation

We extensively benchmarked our framework and compare the performance with prior and concurrent work. See § C.1 for more details in the context of generic 2PC.

Neural Networks We use the CryptoNets [16] and MiniONN [33] networks for the MNIST [30] dataset to compare the performance of our neural network building blocks with generic 2PC of hybrid circuits generated by HyCC [9]. In almost every case, our dedicated building blocks outperform the generic 2PC implementations of our framework, ABY [13], and MOTION [7]. Detailed benchmark results are given in § C.2.

To examine the performance on larger neural networks, we use the MiniONN [33] neural network for the CIFAR-10 [29] dataset, and compare the online run-times using our neural network building blocks with results for GAZELLE [24], DELPHI [35], and CrypTFlow2 [41].[2] The results (see § C.2) show that the recently published CrypTFlow2 offers clearly the best performance, but the performance with our building blocks is comparable to that of GAZELLE and DELPHI.

ABY vs. ABY2.0 According to our experiments, SIMD seems to be *even more important* for a good performance with the ABY2.0 secret-sharing-based protocols than for GMW. Without SIMD, the function-dependent setup phase is a clear drawback of the ABY2.0 sharings, and can result in significantly slower setup phases compared to GMW. With SIMD, the differences in the setup phase diminish. Sometimes, the measured setup run-time was even lower for ABY2.0 compared to the ABY protocols.

For the evaluation of circuits with the generic 2PC implementation, there is no clear winner in the online phase: In most of our experiments, the ABY protocols are better in some settings, and the ABY2.0 protocols are better in other settings. An exception is the CryptoNets [16] benchmark, where the ABY2.0 protocols perform almost always worse than the ABY protocols in the online phase.

When evaluating a neural network using our specialized building blocks, the ABY2.0 protocols are the best choice regarding the online run-times. This is observed best when benchmarking ReLU operations and the two MiniONN [33] networks.

[1] ONNX: `https://github.com/onnx/onnx`

[2] The latter results are taken from [24] and [41], and have been obtained in a different experimental setup and with different bit sizes.

Acknowledgments

This work is supported by the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreements No. 803096 (SPEC) and No. 850990 (PSOTI), and the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM). It was co-funded by the Deutsche Forschungsgemeinschaft (DFG) — SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230, and by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within ATHENE.

References

[1] V. A. Abril, P. Maene, N. Mertens, D. Sijacic, and N. Smart. *'Bristol Fashion' MPC Circuits*. 2019. URL: `https://homes.esat.kuleuven.be/~nsmart/MPC/`.

[2] A. Aly, K. Cong, K. Koch, M. Keller, D. Rotaru, O. Scherer, P. Scholl, N. P. Smart, T. Tanguy, and T. Wood. *SCALE-MAMBA Software*. `https://homes.esat.kuleuven.be/~nsmart/SCALE/`. 2018.

[3] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. *More Efficient Oblivious Transfer and Extensions for Faster Secure Computation*. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS 2013* (Berlin, Germany, Nov. 4–8, 2013). ACM, 2013.

[4] M. Barni, C. Orlandi, and A. Piva. *A privacy-preserving protocol for neural-network-based computation*. In: *Proceedings of the 8th workshop on Multimedia & Security, MM&Sec 2006* (Geneva, Switzerland, Sept. 26–27, 2006). ACM, 2006.

[5] D. Beaver. *Efficient Multiparty Protocols Using Circuit Randomization*. In: *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Proceedings* (Santa Barbara, CA, USA, Aug. 11–15, 1991). Springer, 1992.

[6] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. *Efficient Garbling from a Fixed-Key Blockcipher*. In: *2013 IEEE Symposium on Security and Privacy, S&P 2013* (Berkeley, CA, USA, May 19–22, 2013). IEEE, 2013.

[7] L. Braun, D. Demmler, T. Schneider, and O. Tkachenko. *MOTION – A Framework for Mixed-Protocol Multi-Party Computation*. 2020. IACR Cryptology ePrint Archive: `2020/1137`.

[8] J. Brickell, D. E. Porter, V. Shmatikov, and E. Witchel. *Privacy-preserving remote diagnostics*. In: *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007* (Alexandria, VA, USA, Oct. 28–31, 2007). ACM, 2007.

[9] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider. *HyCC: Compilation of Hybrid Protocols for Practical Secure Computation*. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018* (Toronto, ON, Canada, Oct. 15–19, 2018). ACM, 2018.

[10] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi. *EzPC: Programmable and Efficient Secure Two-Party Computation for Machine Learning*. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2019* (Stockholm, Sweden, June 17–19, 2019). IEEE, 2019.

[11] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. *Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation*. In: *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, Proceedings* (New York, NY, USA, Mar. 4–7, 2006). Springer, 2006.

[12] D. Demmler, G. Dessouky, F. Koushanfar, A. Sadeghi, T. Schneider, and S. Zeitouni. *Automated Synthesis of Optimized Circuits for Secure Computation*. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS 2015* (Denver, CO, USA, Oct. 12–16, 2015). ACM, 2015.

[13] D. Demmler, T. Schneider, and M. Zohner. *ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation*. In: *22nd Annual Network and Distributed System Security Symposium, NDSS 2015* (San diego, CA, USA, Feb. 8–11, 2015). The Internet Society, 2015.

[14] W. Du and M. J. Atallah. *Secure Multi-Party Computation Problems and Their Applications: A Review and Open Problems*. In: *Proceedings of the New Security Paradigms Workshop 2001* (Cloudcroft, NM, USA, Sept. 10–13, 2001). ACM, 2001.

[15] S. Even, O. Goldreich, and A. Lempel. *A Randomized Protocol for Signing Contracts*. In: *Advances in Cryptology: Proceedings of CRYPTO '82* (Santa Barbara, CA, USA, Aug. 23–25, 1982). Plenum Press, New York, 1983.

[16] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. E. Lauter, M. Naehrig, and J. Wernsing. *CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy*. In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016* (New York City, NY, USA, June 19–24, 2016). Vol. 48. JMLR Workshop and Conference Proceedings. JMLR.org, 2016.

[17] N. Gilboa. *Two Party RSA Key Generation*. In: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Proceedings* (Santa Barbara, CA, USA, Aug. 15–19, 1999). Springer, 1999.

[18] O. Goldreich, S. Micali, and A. Wigderson. *How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority*. In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, STOC 1987* (New York, NY, USA, May 25–27, 1987). ACM, 1987.

[19] I. J. Goodfellow, Y. Bengio, and A. C. Courville. *Deep Learning*. MIT Press, 2016. URL: `http://www.deeplearningbook.org/`.

[20] C. Guo, J. Katz, X. Wang, and Y. Yu. *Efficient and Secure Multiparty Computation from Fixed-Key Block Ciphers*. In: *2020 IEEE Symposium on Security and Privacy, S&P 2020* (San Francisco, CA, USA, May 18, 2020–May 21, 2019). IEEE, 2020.

[21] C. Hazay and Y. Lindell. *Efficient Secure Two-Party Protocols. Techniques and Constructions*. Springer, 2010.

[22] W. Henecka, S. Kögl, A. Sadeghi, T. Schneider, and I. Wehrenberg. *TASTY: tool for automating secure two-party computations*. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010* (Chicago, IL, USA, Oct. 4–8, 2010). ACM, 2010.

[23] S. Jha, L. Kruger, and P. D. McDaniel. *Privacy Preserving Clustering*. In: *Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Proceedings* (Milan, Italy, Sept. 12–14, 2005). Springer, 2005.

[24] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan. *GAZELLE: A Low Latency Framework for Secure Neural Network Inference*. In: *27th USENIX Security Symposium, USENIX Security 2018* (Baltimore, MD, USA, Aug. 15–17, 2018). USENIX Association, 2018.

[25] S. Kamara, P. Mohassel, and M. Raykova. *Outsourcing Multi-Party Computation*. 2011. IACR Cryptology ePrint Archive: `2011/272`.

[26] M. Keller. *MP-SPDZ: A Versatile Framework for Multi-Party Computation*. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS 2020* (virtual conference, Nov. 9, 2020–Nov. 13, 2019). ACM, 2020.

[27] V. Kolesnikov and T. Schneider. *Improved Garbled Circuit: Free XOR Gates and Applications*. In: *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008. Proceedings, Part II* (Reykjavik, Iceland, July 7–11, 2008). Springer, 2008.

[28] O. Kowalke. *Boost.Fiber*. Version 1.74.0. Library Documentation. 2020. URL: `https://www.boost.org/doc/libs/1_74_0/libs/fiber/`.

[29] A. Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. `https://www.cs.utoronto.ca/~kriz/learning-features-2009-TR.pdf`. 2009.

[30] Y. LeCun, C. Cortes, and C. J. C. Burges. *The MNIST Database of Handwritten Digits*. 1998. URL: `http://yann.lecun.com/exdb/mnist/`.

[31] Y. Lindell and B. Pinkas. *A Proof of Security of Yao's Protocol for Two-Party Computation*. In: *Journal of Cryptology* 22.2 (2009).

[32] Y. Lindell and B. Pinkas. *Privacy Preserving Data Mining*. In: *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Proceedings* (Santa Barbara, CA, USA, Aug. 20–24, 2000). Springer, 2000.

[33] J. Liu, M. Juuti, Y. Lu, and N. Asokan. *Oblivious Neural Network Predictions via MiniONN Transformations*. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017* (Dallas, TX, USA, Oct. 30–Nov. 3, 2017). ACM, 2017.

[34] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. *Fairplay - Secure Two-Party Computation System*. In: *Proceedings of the 13th USENIX Security Symposium* (San Diego, CA, USA, Aug. 9–13, 2004). USENIX, 2004.

[35] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. *Delphi: A Cryptographic Inference Service for Neural Networks*. In: *29th USENIX Security Symposium, USENIX Security 2020* (Boston, MA, USA, Aug. 12–14, 2020). USENIX Association, 2020.

[36] P. Mohassel and Y. Zhang. *SecureML: A System for Scalable Privacy-Preserving Machine Learning*. In: *2017 IEEE Symposium on Security and Privacy, S&P* (San Jose, CA, USA, May 22–26, 2017). IEEE, 2017.

[37] ONNX Project Contributors. *Open Neural Network Exchange (ONNX)*. 2019. URL: `https://onnx.ai`.

[38] A. Patra, T. Schneider, A. Suresh, and H. Yalame. *ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation*. In: *30th USENIX Security Symposium, USENIX Security 2021* (Virtual Event, Aug. 13–15, 2021). USENIX Association, 2021.

[39] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. *Secure Two-Party Computation Is Practical*. In: *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security* (Tokyo, Japan, Dec. 6–10, 2009). Springer, 2009.

[41] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma. *CrypTFlow2: Practical 2-Party Secure Inference*. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS 2020* (virtual conference, Nov. 9, 2020–Nov. 13, 2019). ACM, 2020.

[42] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar. *XONN: XNOR-based Oblivious Deep Neural Network Inference*. In: *28th USENIX Security Symposium, USENIX Security 2019* (Santa Clara, CA, USA, Aug. 14–16, 2019). USENIX Association, 2019.

[43] M. Rosulek and L. Roy. *Three Halves Make a Whole? Beating the Half-Gates Lower Bound for Garbled Circuits*. In: *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part I*. Springer, 2021, pp. 94–124.

[44] S. Tillich and N. Smart. *(Bristol Format) Circuits of Basic Functions Suitable For MPC and FHE*. 2014. URL: `https://homes.esat.kuleuven.be/~nsmart/MPC/old-circuits.html`.

[45] X. Wang, A. J. Malozemoff, and J. Katz. *EMP-toolkit: Efficient MultiParty computation toolkit*. `https://github.com/emp-toolkit`. 2016.

[46] A. C.-C. Yao. *How to Generate and Exchange Secrets*. In: *27th Annual Symposium on Foundations of Computer Science, FOCS 1986* (Toronto, Canada, Oct. 27–29, 1986). IEEE Computer Society, 1986.

[47] A. C.-C. Yao. *Protocols for Secure Computation*. In: *23th Annual Symposium on Foundations of Computer Science, FOCS 1982* (Chicago, IL, USA, Nov. 3–5, 1982). IEEE Computer Society, 1982.

[48] S. Zahur, M. Rosulek, and D. Evans. *Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates*. In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Proceedings, Part II* (Sofia, Bulgaria, Apr. 26–30, 2015). Springer, 2015.

Appendix A Protocols

Here, we give a high-level overview of the protocols covered and implemented in this work. The protocols are secure in the semi-honest security setting, i.e., corrupted parties follow the protocol, but try to learn additional information. We split the protocols into a setup and an online phase. The former is independent of the parties' inputs and can be precomputed before the actual computation starts.

A.1 Auxiliary Protocols

Secret Sharing Secret sharing denotes a technique to split a secret into multiple shares such that it can be reconstructed from certain subsets of the shares. Important in our case is additive secret sharing over $\mathbb{Z}_{2^\ell}$: To share $x \in \mathbb{Z}_{2^\ell}$, a dealer samples $x_1, \ldots, x_N \in_R \mathbb{Z}_{2^\ell}$ uniformly at random such that $x = x_1 + \cdots + x_N$, and distributes the x_i among the parties. This is denoted as $\langle x \rangle^A = (x_1, \ldots, x_N)$. Given all shares the original secret can be recovered.

Oblivious Transfer Oblivious transfer (OT) [15] is a two-party protocol between a sender and a receiver. The sender inputs two messages m_0, m_1 and the receiver inputs a bit b. Then the receiver obtains m_b without learning m_{1-b} and the sender does not learn anything about b. There exist many variants of OT, e.g., correlated OT (C-OT) where the messages are chosen randomly under some correlation supplied by the sender [3].

Multiplication Protocols C-OT can be seen as computing a secret sharing of the product of two private bits. This has been extended to multiplication of integers, and generalized to products of vectors and matrices [17, 13, 36]. Moreover, two multiplication with private inputs can be combined to obtain a multiplication with two *shared* inputs. The arithmetic protocols covered below make heavy use of such OT-based multiplications.

A.2 General-Purpose 2PC

General-purpose 2PC protocols allow us to securely evaluate functionalities encoded in the form of Boolean or arithmetic circuits. Here we use and combine five existing protocols (denoted with Y, A, B, α, and β) with different properties.

Yao's Protocol (Y) This is a 2PC protocol for Boolean circuits [47, 46]. One party, the *garbler*, creates an encoding of the circuit, the *garbled circuit*, and sends it to the second party, the *evaluator*. Given an encoding of the circuit inputs, the latter is able to obliviously evaluate circuit to obtain an encoded output. This can be decoded with help of decoding information generated by the garbler.

To this end, the garbler creates for each circuit wire w two keys k_w^0, k_w^1. For each gate g with input wires a, b and output wire c it creates a garbled table where for each possible combination of input values $\alpha, \beta \in \{0, 1\}$ the keys k_a^α and k_b^β are used to encrypt $k_c^{g(\alpha,\beta)}$. During evaluation the evaluator obtains k_w^b if the wire holds value b (denoted as $\langle b \rangle^Y$). Hence, given one key for each circuit inputs, the evaluator can evaluate the whole circuit while only seeing random-looking keys instead of the plain values. We implement Yao's protocol with then state-of-the-art optimizations such as FreeXOR [27], fixed-key AES [6, 20], and Half-gates [48].[3]

GMW (A/B) The GMW protocol [18] is a generic MPC protocol for Boolean (B) and arithmetic (A) circuits in the full-threshold, semi-honest security setting. We consider GMW over the ring $\mathbb{Z}_{2^\ell}$, which for $\ell = 1$ is equivalent to bits $\{0, 1\}$ with $\oplus$ and $\wedge$. The following description uses arithmetic notation, but unless stated otherwise holds also for the Boolean case.

A value $x \in \mathbb{Z}_{2^\ell}$ is shared using additive secret sharing $\langle x \rangle^A$. In the Boolean domain this is denoted as $\langle x \rangle^B$. Since the secret sharing scheme is linearly homomorphic, we can compute the sum of shared values by adding the shares locally. A multiplication $\langle z \rangle^A \leftarrow \langle x \rangle^A \cdot \langle y \rangle^A$ on the other hand requires interaction among the parties. As outlined in § A.1, shared values can be multiplied e.g. with OT-based multiplication protocols. Here, we use these to precompute multiplication triples (MTs) [5], i.e., random shared triples $(\langle a \rangle^A, \langle b \rangle^A, \langle c \rangle^A)$ such that $ab = c$, in the setup phase. Then, during the online phase the inputs x, y are masked with a, b from the MT, and then reconstructed. Finally a sharing of z can be computed using a linear combination of shared values with public coefficients. We also support mixed products of a bit $\langle b \rangle^B$ and a number $\langle n \rangle^A$.

ABY2.0 Sharing (α/β) Recently, ABY2.0 [38] was published as a successor protocol suite to the original ABY protocols [13]. They also combine Yao's protocol (Y) with a Boolean and an arithmetic secret-sharing-based protocol. Instead of using GMW for the latter, they designed a new protocols which "uses a different perspective of [Beaver's circuit randomization] technique" [38]. Furthermore, they designed a new set of conversion protocols and building blocks for various applications. To distinguish the new protocols, we use the term *ABY2.0 sharing* in this work and denote them with α and β.

To share a value $x \in \mathbb{Z}_{2^\ell}$, it is masked with a random value $\Delta_x \leftarrow x + \delta_x$ and the mask is additively secret shared among the parties: $\langle x \rangle^\alpha = (\Delta_x; \langle \delta_x \rangle^A)$. Note that δ_x can be generated independently of x during the setup phase. Addition works locally by adding the shares. For multiplications $\langle z \rangle^\alpha \leftarrow \langle x \rangle^\alpha \cdot \langle y \rangle^\alpha$, we need to compute $\langle \delta_x \cdot \delta_y \rangle^A \leftarrow \langle \delta_x \rangle^A \cdot \langle \delta_y \rangle^A$ in the setup phase. Then, in the online phase we can locally compute $\langle z \rangle^A$, and convert this to $\langle z \rangle^\alpha$ by masking it with δ_z and reconstructing the $\Delta_z = z + \delta_z$. Compared to GMW, only one reconstruction is needed, which halves the communication during the online phase.

We also support special products between shared bits and numbers $\langle b \cdot n \rangle^\alpha \leftarrow \langle b \rangle^\beta \cdot \langle n \rangle^\alpha$ and $\langle b_1 \cdot b_2 \rangle^\alpha \leftarrow \langle b_1 \rangle^\beta \cdot \langle b_1 \rangle^\beta$ and *improve* the setup phase compared to [38].

Comparison The three protocols have different properties: Yao's protocol has a constant round complexity, whereas the other protocols need one round of interaction for each non-linear layer of the circuit. On the other hand transferring the garbled circuit needs a larger amount of communication. Hence, which approach performs better depends on the available network bandwidth and latency. For a more detailed comparison of the ABY and ABY2.0 protocol suites, see § 2.

Conversion Protocols To combine the different protocols and exploit their respective advantages, we provide conversion protocols between the five kinds of sharings. Figure 1 gives an overview of

[3] This was considered state-of-the-art until the recently published [43].

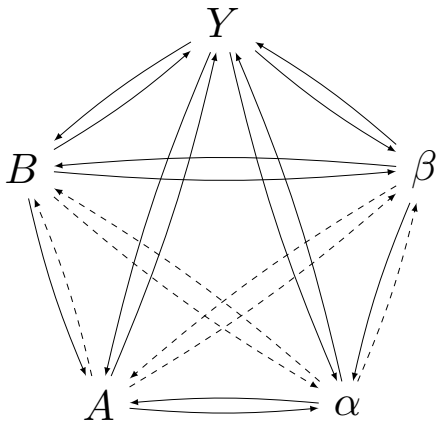

Figure 1: Overview of the protocol conversions. Dashed lines denote a conversion via a third protocol.

the conversions. Some of them are implemented as a composition of two other protocols, e.g., for $A \to B$ we take the path $A \to Y \to B$.

A.3 Building Blocks for Neural Networks

Neural networks often have a very regular structure. The input is a tensor, a multi-dimensional array, which gets transformed by a sequence of different layers. Common are linear and pooling layers combined with (non-linear) activation functions. [19] We use the generic protocols (§ A.2) to construct building blocks to securely evaluate these layers on shared inputs. Since some are more efficiently computed with an arithmetic protocol and others with a Boolean protocol, we use the conversions (§ A.2) to change the representation of the data as necessary.

Fixed-Point Arithmetic Parameters, inputs, and outputs of neural networks are usually represented by floating-point numbers. While Boolean MPC can be used to evaluate Boolean circuits encoding the floating-point operations [12], the native operations of arithmetic MPC protocols are usually more efficient. Therefore, we encode decimal numbers into elements of $\mathbb{Z}_{2^\ell}$ using a fixed-point representation and the truncation protocol by [36] for A-shared values. It involves only local computation which makes it very efficient. The downside is that this protocol does not achieve perfect correctness and errors can be introduced.

Linear Layers For fully-connected, and convolution layers – both are essentially matrix multiplications, we use the arithmetic sharings A and α. While we can build matrix multiplication circuits out of additions and multiplications, it is more efficient to operate directly on (element-wise shared) matrices. The multiplication in both sharings generalize naturally to the multiplication of (element-wise shared) matrices: In GMW (A), we use multiplication triples $(\langle \mathbf{A} \rangle^A, \langle \mathbf{B} \rangle^A, \langle \mathbf{C} \rangle^A)$ where each element is a matrix. Then secure matrix multiplication works exactly as integer multiplication, but using element-wise addition and matrix multiplication instead of integer addition and multiplication, respectively [36]. This way, we mask each entry only once, instead of doing it separately for each integer multiplication, which saves communication. In the same fashion, the integer multiplication in the α sharing, can be generalized to matrix multiplication [38]. Then, $\Delta_{\mathbf{X}}$, $\delta_{\mathbf{X}}$ etc. are also matrices of appropriate size.

ReLU The ReLU operation ($\mathrm{ReLU}(x) = \max(0, x)$) is a commonly used activation function. There are different approaches to compute a ReLU layer. First, note that, since the most significant bit $\mathrm{msb}(x)$ encodes the sign of a number x in twos-complement, one can write $\mathrm{ReLU}(x) = \neg\mathrm{msb}(x) \cdot x$. Thus, the problem can be reduced to obtaining $\mathrm{msb}(x)$ and computing the product. The MSB can be

computed either by converting the whole share into a Boolean sharing or via specialized bit extraction protocols (e.g. [11]).

MaxPool To evaluate a MaxPool layer, we use a Boolean sharing (Y, B, or β) because of the required comparisons. For each position of the window, the maximum is computed as follows: The circuit consists of a balanced binary tree where each node is comprised of a $>$ comparison circuit connected to a multiplexer such that the maximum of the input values is forwarded towards the root of the tree. For the comparison, either size-optimized (for Y) or depth-optimized (for B, β) circuits are used.

AvgPool AveragePool is a linear operation, consisting of a summation and an element-wise division through the window (or kernel) size k. Since the window size k is part of the functionality, and, thus, publicly known, we can write the latter as a multiplication with the constant $1/k$. However, since $1/k < 1$ for all non-trivial cases, we need to take the fixed-point representation into account, and execute a truncation protocol.

Appendix B Extending the MOTION Framework

This work builds upon and extends the MOTION framework for mixed-protocol multi-party computation [7]. For more information about the original architecture of the framework, the reader is referred to [7, § 4]. In the following we give a short overview of our extensions.

Communication The communication subsystem of the MOTION framework was *redesigned* to allow easy-to-use and flexible asynchronous message-based communication. The challenge is that multiple messages might concurrently be sent and expected on the sender and the receiver side, respectively. So the system needs to make sure that every message ends up in the right place without being able to rely on an order among the messages. Also – in the multi-party setting – messages may be sent to and received from multiple parties. A low-level *transport* implements the actual sending and receiving of messages between two parties, e.g., via TCP. The details of connection setup and use as well as the used libraries are completely hidden from the user. Thus, the framework can be easily adapted to use other transports instead, e.g., WebSockets or QUIC. The high-level `CommunicationLayer` offers a simple but flexible API for sending messages to other parties. The methods return immediately while the sending itself happens in the background. Received messages are passed to message handlers which define how incoming messages of certain types are processed. For synchronization between all parties, we provide a builtin *barrier* as synchronization mechanism.

Protocol Implementation Based on the terminology of circuits, two of the main concepts in the MOTION framework [7] are *gates* and *wires*. Both are abstract interfaces which have been redesigned to provide a cleaner API and reduced memory overhead. Wires are the passive components and hold the local share of a secret-shared value. They can be seen as low-level variables and have a builtin synchronization mechanism allowing a consumer to safely wait for it to obtain its value. A gate object represents the active part of the computation encapsulating the protocol for a single operation, e.g., a primitive operation in a circuit or a tensor operation in a neural network. We introduce so-called *protocol providers*, which bundle all the required functionality for a 2PC protocol with a common interface. Hence, circuits can be constructed in a completely generic way: Given wires and an identifier of an operation, a protocol provider will construct the corresponding gate and return the output wire.

Backend MOTION [7] gathers all required components in so-called backend classes. The original `Backend` in MOTION [7] contained a lot of unrelated functionality and was tightly coupled with the rest of the framework. Now, the responsibility of the new backend classes has been reduced such that they primarily construct and coordinate the required components (e.g., provider of OTs and protocols) for their use-cases. Depending on the setting, different implementations for certain protocol can be

chosen. Currently, there are backends for the two-party and the multi-party settings, as well as a specialized backend for evaluating tensor operations in neural networks.

Execution The encapsulation of the primitive operation in gate objects allows the decoupling of the protocols description and its evaluation strategy. MOTION [7] uses *fibers* to evaluate gates, i.e., threads that run in user space opposed to standard threads managed by the operating system kernel. They need less resources, and switching between fibers is done completely in user space. This offers additional flexibility, e.g., allows using custom schedulers and allocators, and can yield improved performance compared to threads [28]. We extracted the execution code into *executor* classes, that takes a collection of gates and executes them according to some strategy, so that different strategies can be implemented in different executors. As in [7], the executor for general-purpose MPC with arbitrary circuits creates fibers for all gates which are then executed by a thread pool. For neural networks, we use a different approach: Their computation graphs are often very simple or even straight-line, and the tensor operations are more complex than the primitive operations in circuits. Hence, we evaluate the gates sequentially while parallelizing the gate evaluations themselves.

Statistics MOTION [7] records detailed run-times during the execution, and computes the mean, median, and standard deviation for repeated experiments. We added support to output this data in the JSON format together with communication statistics and metadata of the experiments (e.g., experiment name, hostname, command line arguments, etc.). This makes it easy to import the data in other software for further analysis.

Implemented Protocols MOTION [7] was released with three protocols for generic MPC with an arbitrary number of parties. This work *additionally* implements support for the five 2PC protocols discussed in § A.2. The original OT implementation by [7] was revised and extended, e.g., with vectorization for C-OT (cf. [36]), and the OT-based multiplication protocols (§ A.1) have been implemented on top. In addition to the general-purpose protocols, specialized building blocks (§ 2, § A.3) have been implemented for the most common operations in neural networks. Depending on the concrete operations, they are implemented using either the arithmetic or the Boolean 2PC protocols.

File Formats MOTION [7] allows building applications using its C++ interface and can also import circuits from various simple file formats used in the MPC community [44, 1, 13]. So far, HyCC [9] circuits were only rudimentary supported via a modified version of the HyCC adapter for ABY by [9]. A new `HyCCAdapter` has been developed that uses HyCC's libcircuit to convert the HyCC circuits into the respective MOTION data structures while hiding this from the MOTION user. Moreover, a new `OnnxAdapter` enables support for neural network descriptions provided in the ONNX file format [37], an open standard for deep learning models. With the ONNX library[4], the adapter can parse the description and automatically construct the respective tensor operations in MOTION. Both new adapters can be optionally enabled at compile time of the MOTION framework if the respective libraries are available.

Appendix C Performance Evaluation

C.1 Generic 2PC

Garbling Engine We measure the raw garbling and evaluation performance of our implementation of the half-gate [48] garbling scheme with fixed-key AES [6, 20] and AES-NI, and benchmark it also with various circuits of different complexity such as AES-128, SHA-256, ReLU, and comparisons. It achieves a comparable performance to the EMP-Toolkit [45], and is $1.8\times$ faster when SIMD operations are enabled. The garbling rate of a single thread is sufficient to saturate a $10\,\mathrm{Gbit/s}$ network links.

[4] ONNX: `https://github.com/onnx/onnx`

Boolean and Hybrid Circuits We use AES-128 and SHA-256, to benchmark the performance of the Boolean protocols B, β, and Y. Moreover, we use the hybrid circuits for biometric matching generated by HyCC [9], which are evaluated using a combination of an arithmetic and a Boolean protocol. In both cases, we compare with the ABY framework [13] and the generic N-party protocols implemented in MOTION [7]. The overall performances are similar. Without SIMD, ABY [13] always has the best run-times, which can be explained by the worse parallelization used in the MOTION framework [7] (see below). With SIMD the performance of MOTION (including this work) becomes competitive (up to $20\times$ better than ABY during the online phase in the LAN). SIMD is not supported by the ABY backend for HyCC [9].

Multi-Threading in MOTION MOTION [7] uses fibers to evaluate the gates of a circuit. However, it had not yet been examined how efficient this approach to parallelization is. We benchmark this by evaluating the AES-128 circuit with the Boolean protocols. Regardless of the number of threads the speedup stays below 2 for Yao's protocol (Y). For the B and β protocols, we cannot see any significant speedup. This is likely because Yao's protocol uses cryptographic operations, whereas the other protocols use only cheaper bit-wise operations in the gates. We conclude that fibers are a good and easy-to-use method for implementing more complex protocols that need more computation and (rounds of) interaction. They are also helpful for prototyping new protocols, but they are *not* an efficient parallelization scheme for circuits built of cheap primitive operations. In such cases, grouping the gates in layers and evaluating these multi-threaded is more advantageous (cf. [13]).

C.2 Neural Networks

Tables 1 and 2 contain our benchmark results for the CryptoNets [16] and MiniONN [33] neural networks for the MNIST dataset [30]. We compare our results with ABY [13] and MOTION [7]. In Table 3, we compare the online run-times for the MiniONN [33] neural network for the CIFAR-10 dataset [29] with GAZELLE [24], DELPHI [35], and CrypTFlow2 [41].

Table 1: Run-times in ms for evaluating the CryptoNets [16] neural network with ReLU using numbers of bitlength $\ell = 32$. This work with NN Ops uses specialized neural network operations (§ A.3), whereas all other categories use a hybrid circuit generated using HyCC [9]. With SIMD, 16 copies of the network are evaluated in parallel. All protocols were executed with $N = 2$ parties, and the best run-times are marked in bold.

Implementation	Protocol	LAN		WAN	
		Setup	Online	Setup	Online
ABY [13]	$A + B$	396.3	123.0	2 613.6	607.6
	$A + Y$	393.1	128.6	2 450.1	572.7
MOTION [7]	$A + B$	1 970.9	1 457.8	4 785.8	2 173.1
	$A + Y$	2 017.3	1 453.6	4 791.4	2 211.9
MOTION [7] w/ SIMD	$A + B$	916.9	101.4	2 889.7	279.8
	$A + Y$	897.2	96.8	2 935.9	248.7
this work	$A + B$	2 221.6	2 075.4	5 961.3	3 059.0
	$A + Y$	1 962.0	2 177.4	5 474.3	2 805.8
	$\alpha + \beta$	3 626.0	3 241.0	7 257.0	3 226.3
	$\alpha + Y$	3 525.6	3 261.0	7 400.3	3 371.6
this work w/ SIMD	$A + B$	650.9	145.3	2 810.5	263.3
	$A + Y$	645.5	142.2	2 777.3	**223.7**
	$\alpha + \beta$	572.8	202.3	2 807.6	251.9
	$\alpha + Y$	590.1	210.6	2 667.3	254.4
this work w/ NN Ops.	$A + B$	111.2	11.1	**1 005.7**	1 047.4
	$A + Y$	**103.4**	11.1	1 062.4	858.6
	$\alpha + \beta$	129.1	28.1	1 766.7	982.2
	$\alpha + Y$	136.6	**8.7**	1 539.4	858.6

Table 2: Run-times in ms for evaluating the MiniONN [33] MNIST neural network using numbers of bitlength $\ell = 32$. This work with NN Ops uses specialized neural network operations (§ A.3), whereas all other categories use a hybrid circuit generated using HyCC [9]. With SIMD, 16 copies of the network are evaluated in parallel. All protocols were executed with $N = 2$ parties, and the best run-times are marked in bold.

Implementation	Protocol	LAN		WAN	
		Setup	Online	Setup	Online
ABY [13]	$A + B$	1 516.3	2 997.2	4 903.5	8 164.1
	$A + Y$	1 559.0	1 453.5	4 960.0	4 194.1
this work	$A + B$	6 423.3	49 712.7	10 391.8	46 298.3
	$A + Y$	16 138.6	15 073.4	19 085.8	16 387.4
	$\alpha + \beta$	109 778.9	48 342.2	110 201.4	48 562.7
	$\alpha + Y$	15 939.9	14 726.1	19 235.7	15 867.0
this work w/ SIMD	$A + B$	1 239.6	3 037.3	5 163.2	2 903.6
	$A + Y$	1 514.1	936.8	6 129.0	1 467.3
	$\alpha + \beta$	7 592.7	3 324.8	9 330.1	2 954.8
	$\alpha + Y$	1 315.6	928.6	6 664.4	**973.1**
this work w/ NN Ops.	$A + B$	649.7	171.5	4 826.6	5 138.1
	$A + Y$	**523.8**	126.7	**4 476.5**	2 574.9
	$\alpha + \beta$	737.0	151.6	6 424.5	4 654.0
	$\alpha + Y$	676.1	**81.8**	5 908.9	1 235.7

Table 3: Online Run-times in ms for evaluating MiniONN [33] CIFAR-10 neural network with ℓ-bit numbers. The run-time of GAZELLE is taken from the corresponding publication [24], and the numbers for DELPHI [35] and CrypTFlow2 are taken from the CrypTFlow2 paper [41]. Note that they were obtained in a different experimental environment. All protocols were executed with $N = 2$ parties. The best run-times are marked in bold, although different bit sizes ℓ are used by the different implementations.

Implementation	Protocol	ℓ	LAN	WAN
GAZELLE [24]	$A + Y$	60	3 560.0	—
DELPHI [35]	$A + Y$	41	≈4 070.0	—
CrypTFlow2 [41]	$A + OT$	41	≈ **420.0**	—
this work w/ NN Ops	$A + B$	32	1 167.7	20 773.1
	$A + Y$	32	1 246.8	34 783.5
	$\alpha + \beta$	32	**624.2**	7 882.5
	$\alpha + Y$	32	695.6	**6 756.2**
	$A + B$	64	2 024.6	37 584.9
	$A + Y$	64	2 231.1	66 187.8
	$\alpha + \beta$	64	**989.5**	11 768.9
	$\alpha + Y$	64	1 226.6	**10 591.8**