

Optimization for Secure Two-Party Inference with Homomorphic Encryption

Christopher Knop, Christopher Harth-Kitzerow*, Nina Schwanke*

**Chair of Network Architectures and Services*

School of Computation, Information and Technology, Technical University of Munich, Germany

Email: christopher.knop@tum.de, christopher.harth-kitzerow@tum.de, nina.schwanke@tum.de

Abstract—In this paper, we build upon a previous project where we used Homomorphic Encryption (HE) to improve non-linear and linear operations for Secure Two-Party Computation (2PC) and integrated them into a Secure Multi-Party Computation (MPC)-framework. We used Beaver triples [1] to put most computation in the offline phase. However, we realized that our algorithms require more Boolean and arithmetic triples than anticipated. Hence, we decided to install further optimizations, especially in the LAN setting. We optimized ReLU, Convolution, and Batch Normalization. We show that our optimizations improve performance, particularly for LAN. On top of that, only about 14.89% of our computation is done online, which significantly improves on the implementations of alternative frameworks, such as SecONNs, which have about 65.77% of the total computation in the online phase.

Index Terms—secure multiparty computation, homomorphic encryption, optimization, benchmark

1. Introduction

Secure Multi-Party Computation (MPC) solves the problem of collaboratively computing any function while also preserving the privacy of the parties' inputs. This is often required for private communication as for instance, in satellite communications [2]. As neural networks have gained widespread adoption, they can be used for private inference as for example, for cancer detection on confidential data. This way, the patients' images and the trained network remain private [3]. In this paper, we examine protocols that use Homomorphic Encryption (HE) for MPC and aim to optimize communication and end-to-end times.

MPC enables multiple parties to collaboratively evaluate a function while preserving the privacy of their respective inputs. This capability is essential in scenarios where sensitive information must be processed jointly without disclosure. For example, MPC can facilitate secure communication and coordination between satellites while protecting confidential operational data [2]. Another important application is privacy-preserving neural network inference, where medical images can be analyzed for tasks such as cancer detection without revealing either the patients' data or the model parameters [3]. In this paper, we investigate MPC protocols based on HE and explore optimizations to reduce communication overhead and end-to-end execution time.

This work extends a prior implementation effort based on OpenCheetah [4], an open-source framework that leverages HE to optimize Secure Two-Party Computation (2PC)

operations for MPC. However, we found that additional optimizations were necessary to minimize online execution time while maintaining a reasonable offline time in both LAN and WAN environments. Here, the offline phase refers to all computation that can be done without any input. Only in the online phase do both parties have to provide their inputs.

First, we will give a short introduction to the topic, give an overview of our optimizations, and finally evaluate the results by comparing our implementation against well-established HE-frameworks OpenCheetah and SecONNs

2. Background

2.1. Secure Two-Party Computation

In general, MPC solves the problem where n parties try to compute a common function, and each party might hold private input. In the special case with only two parties ($n = 2$; commonly abbreviated as 2PC), there are implications that allow for optimizations, as discussed in the following sections. Typical approaches include: a) a trusted third party which is honest and all parties can rely on, b) Oblivious Transfer (OT) (see Section 4.6). However, the latter usually requires more communication for arithmetic operations, as it operates at the bit level [1]. Therefore, HE has recently attracted attention because it reduces communication requirements, but at a higher computational cost, especially in the two-party case, where only one party needs to encrypt their input, while the other can use plaintext-ciphertext operations.

2.2. Homomorphic Encryption

Homomorphic Encryption allows for computation on ciphertexts, such as addition or multiplication, usually in a ring modulo a prime or a power of two [4]. Regarding MPC, previous work has primarily focused on Ring Learning with Errors (RLWE) because its operations are cheap, and with appropriately chosen parameters, the error terms can be circumvented [4]. For the operations OpenCheetah implements, the multiplicative depth is low, and as the layers are independent, the error term does not increase and can be neglected [4].

2.3. Oblivious Transfer

We use OT to generate Boolean triples for non-linear operations. OT solves the problem where one party has a

choice bit b and another party holds two messages m_0, m_1 in the end the party holding b wants to receive message m_b while the other party must not learn b . This must be efficient, as our protocols require several hundred million Boolean triples. OpenCheetah uses the three different implementations, which we briefly discuss in Section 4.6 [4].

2.4. High-Throughput MPC

For our implementations, we used the open-source framework HPMPC, which supports various algorithms and schemes for different security settings in MPC [5]. We focus on the 2PC protocols and optimizing the 2D convolutions, Batch Normalization, and ReLU operations. For this, we defined two protocols denoted AB and $AB2$ (see Appendix A.1). The main difference is that, for $AB2$ party P_0 does not have shares of input B (the filters for convolution see Sec. A.1), which simplifies the process.

3. Related work

This work is based on the projects OpenCheetah [4] and SecONNs [6]. The former is an online-only phase framework that implements several linear and nonlinear operations optimized for 2PC using RLWE. SecONNs optimizes OpenCheetah by using Number Theoretic Transform (NTT) and Random OT (ROT) for ReLU. As a result, they were able to put approximately 20 % of the runtime in the offline phase. We build two 2PC protocols that utilize their algorithms but make use of beaver triples to put most computation in the offline phase.

Other closely related projects include BumbleBee [7] and Secretflow [8], which have further optimized the packing methods by OpenCheetah to reduce the size of ciphertexts relevant for matrix multiplication using HE and Troy-Nova by Liu et al. [9], who implemented SEAL (a HE library by Microsoft) on GPU and implemented the algorithms introduced by OpenCheetah and Bumblebee for 2D-Convolution and matrix-multiplication.

4. Optimizations

In this context, we will often refer to the inputs A and B as images and filters, respectively (see Section A.1), since that is how the inputs are interpreted in the context of convolutional neural networks.

4.1. Security Definitions

Our protocols are secure in the semi-honest adversary setting (also known as the passive setting), where an adversary executes the protocols as defined but may use any information given to them to derive additional information [1]. We are using HE for 128 bit security, hence, an attacker requires at least an order of 2^{128} operations to obtain information [10, 11].

4.2. Implementation

OpenCheetah combines serialization, sending, and deserialization. However, this adds superfluous delays to

our AB protocol (see Section A.1), as it implies that while one party is serializing and sending, the other must wait and can start serializing and sending only after the first party finishes. Profiling has shown that serialization itself can be a bottleneck. Thus, to reduce the receiver's waiting time, we introduce a method that combines sending and receiving by first having both parties serialize their encrypted shares, exchange them, and finally deserialize the received messages simultaneously, thereby minimizing the time each party has to wait.

Additionally, by default, OpenCheetah flushes its packets immediately, which generates unnecessary traffic. Since, when sending a ciphertext, it is also important to include the packet size. Instead of sending the size and shares separately, send them in a single packet. Therefore, we use buffering and only flush when all messages to send have been registered, ensuring everything arrives. This way, we can fill the sending buffers appropriately, maximizing our throughput while minimizing communication.

When performing the benchmarks, we noticed that because OpenCheetah is not perfectly parallelized, it is more efficient to use it in single-threaded mode when introducing a batch size parameter. This is because OpenCheetah assumes the batch size to be equal to one for most operations. This currently comes with the disadvantage that we have to encrypt the input filters multiple times. We note that it is possible to precompute the encryption of the filters once and then share it across all processing threads.

4.3. Initial Setup

As we have to close and reopen connections to perform operations when required by HPMPC. To prevent the two frameworks (HPMPC/OpenCheetah) from interfering with each other and to reduce the number of required ports, we initially had to exchange the public keys and seeds required for OT multiple times. To reduce overhead, we introduced a static singleton that, as soon as it is created, exchanges all required keys and random seeds. This is particularly helpful for Correlated OT (COT), as setting it up takes about one second on average. This improved architecture enables us to repeatedly call the Boolean triple Generator with minimal overhead compared to running it once and generating all the required shares in only a single call. For this, we had to implement additional disconnect and connect methods, as the input-output channels by EMP-OT¹ must be consistent and cannot change. This was not an issue for OpenCheetah, as they do not need to free the ports because they do not have two interfering frameworks. We thus minimize the overhead associated with the connection build-up between the two parties.

4.4. Convolution

Firstly, we introduce a parameter for the batch size for our protocols that allows us to encode our filters once.

Additionally, we used SecONNs optimizations for Convolution and used NTT to encode the filters. Since we only encode the filters once, we can save time for subsequent batches (see Section 5). This optimization

1. OpenCheetah uses this framework for OT

also improves the multiplication operation for HE, saving execution time [6](see Section 5).

4.4.1. GPU

Based on the idea of SecONNs, we also added an option to use a GPU version of SEAL. However, instead of using Troy, we implemented Convolutions using Troy-Nova, which is a more modern version of Troy. Similarly to SecONNs, they optimize for batch sizes greater than one using NTT and only encode the filters once.

Additionally, Troy-Nova supports what they call reverse convolutions, which enables us to encrypt the filters rather than the images. This reduces the communication required for multiple batches, as the filters need to be sent only once, whereas, with the other implementation the images must be sent separately for each batch.

Another main difference between our implementation and SecONNs is that we use the GPU for encoding/decoding and encryption/decryption, whereas SecONNs uses SEAL for these operations and only uses the GPU for the convolution itself [6].

4.5. Batch Normalization

OpenCheetah has two methods for batch normalization and switches between them based on which has the more compact packing. Their first method is just using elementwise multiplication, while their second method makes some optimizations to reduce the size of the packed ciphertexts. However, when first using this, we used only a single thread for their HPMPC implementation. We decided to split the work onto multiple threads, similar to what OpenCheetah does.

Nonetheless, we also fused subsequent batch normalizations and convolutions into a single convolution call. Therefore, we were able to remove all calls to batch normalization, which has only a minor impact on convolutions but greatly improves runtime and communication (see Section 5).

OpenCheetah also supports fused Batch Normalization (BN). However, they still require a couple of calls to BN (see Section 5).

4.6. Arithmetic and Boolean triples

ReLU in MPC consists of two phases: first, solve the millionaire’s problem ($a < b$) and second, multiply the result by the input shares. This fulfills the definition of ReLU (1).

$$\text{ReLU}(x) = \max(x, 0) \quad (1)$$

Initially, we attempted to solve the second phase using HE and implemented it using OpenCheetah’s elementwise multiplication, which it uses for BN. However, even after parallelizing this operation, it still took too long in the WAN setting as it requires about 750 MiB for ResNet50. Which itself given a network with a bandwidth of 200 Mbit s^{-1} takes about 52.5 s. Therefore, we decided to use OpenCheetah’s COT approach instead, which leverages the optimized OT scheme that utilizes seeded randomness to significantly reduce communication [12]. This takes

about half a second longer to compute but drastically reduces communication (see Section 5).

The first phase, however, requires numerous Boolean shares. OpenCheetah supports three schemes to generate these. The first uses ROT. However, this approach, uses random data without taking any input, which makes it hard to use for us, since we have given input shares. A similar problem arises with their second approach, which uses 1-out-of-8 OT, but it also changes the input values. Thus, the only approach we can use without making changes to HPMPC is to use their 1-out-of-16 OT scheme. This requires more communication as it does not utilize COT. However, to further optimize it, we simply spread the workload over multiple threads, which initially did not improve our runtime, as we assumed we would require fewer shares.

4.7. LAN and WAN

First of all, unlike previous work, all our communication is spread across multiple ports and threads. Thus, the ciphertexts can be quickly shared, serialized, and deserialized.

To maximize the time on the CPU and reduce time spent waiting for packets to arrive, we implemented a thread-safe queue that allows use to have two separate threads for sending and receiving the ciphertexts. Here, the send thread waits for the computing threads to encrypt a layer, then pushes them to the send thread, which then sends the number of ciphertexts, the total package size, and all the ciphertexts. The receive thread then receives the packages, deserializes them, and pushes them onto the receive queue for the computing threads to pull from as needed. By using thread-safe queues and using only one thread for sending and receiving, respectively, we ensure that the order in which we send/receive packets does not change without implementing a more complex mechanism. This enables the computing threads to run continuously without waiting for all packets to arrive, but only the ones they need for the next step, which is critical in slower networks or over WAN.

Finally, we added an option to execute only the offline phase and store all generated triples in a file, which can then be used to compute the online phase later. We achieved this by implementing a first-in, first-out queue, which allows us to store multiple offline runs in advance. However, this has not been optimized for multiple runs and may require significant copying when reading the data, as the remaining triples for subsequent runs must be moved to the front after the first set of triples is read and erased. A more efficient way to solve this could be to store a pointer to the start of unread data and delete the file once every byte has been read.

5. Evaluation

We evaluate our implementation by comparing it to OpenCheetah and SecONNs. All evaluations were distributed and parallelized with 32 threads on the same CPU (AMD EPYC 7543). Additionally, we compare the results in LAN and WAN settings. We used OpenCheetah’s *ResNet50* architecture for benchmarking and implemented an equivalent architecture for HPMPC, ensuring that all

TABLE 1: Data sent by P_0 (upper) and P_1 (lower) in MiB using the AB2 protocols.

	Conv-2D	BN	FC	Triples	Total
Cheetah	834.17	162.11	0.72	282.57	1279.57
	167.77	189.91	1.19	282.54	451.50
SecONNds	834.16	162.11	0.72	282.57	1279.56
	167.77	189.91	1.19	282.54	451.50
Our impl.	834.11	0	0.72	378.83	1213.66
(87)	167.77	0	1.19	344.49	513.45
Our impl.	834.11	0	0.72	935.61	1770.44
(187)	167.77	0	1.19	901.80	1070.76
Our impl.	834.11	0	0.72	1741.18	2576.01
(287)	167.77	0	1.19	1707.37	1876.33

input dimensions are identical. HPMPC also supports different adders here denoted as 87 for ripple carry adder, 187 for parallel prefix adder, and 287 for 4-way parallel prefix adder. For this project, the difference between these approaches lies in the number of triples they require, and thus, less communication may be required in the online phase. To evaluate our implementation, we will consider only the server-side perspective (P_0), since the client-side runtimes are nearly identical, as they have to wait for the results of P_0 and would complicate our visualization.

5.1. Communication

Table 1 shows how much data the server and client have to send to one another (P_0 upper value and P_1 lower value). The most interesting operation for communication is ReLU, as its implementation differs significantly from that of other frameworks. We see that using a ripple-carry adder results in the least communication, as it requires the fewest Boolean triples. For all frameworks, we utilized the available option to fuse batch normalization with convolution operations. Nonetheless, OpenCheetah and SecONNds still required certain batch normalization steps. Additionally, we excluded the communication required for exchanging public keys and the data necessary for COT.

We note that the numbers provided for our protocols do not include the online-phase communication, but only the data required for the operations defined in the previous sections. The parties send about 150-200 MB during the online phase.

For our AB protocol, one must sum up the data sent by P_0 and P_1 for FC, BN and 2D-convolutions to determine the total amount each party must send, as both parties now perform the same computation and communication.

5.2. Local Area Network

For LAN, we averaged the runtimes over 10 interferences and display the averages as well as the minimum and maximum values.

Figure 1 compares the end-to-end times. Note that SecONNds does not support optimization to precompute triples in the offline phase for bit length 32 and ResNet50. Hence, the offline time here is just the filter encoding using NTT. This way, about 65.77% of the total computation is done in the online phase. Although our approach with the 4-way parallel-prefix-adder achieves the best online time ($\approx 4.86\%$ online phase), its offline phase lags behind because it requires the most triples and thus has the worst

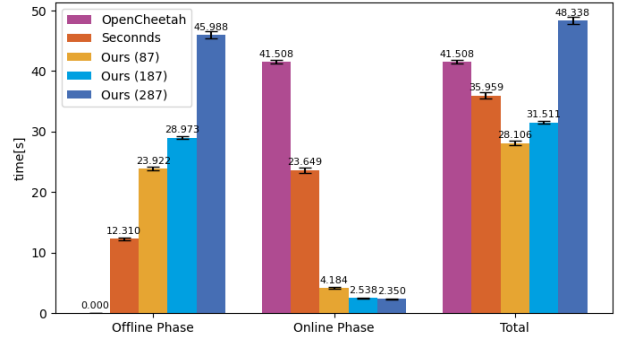


Figure 1: Online and offline times for the different frameworks in seconds (includes NTT).

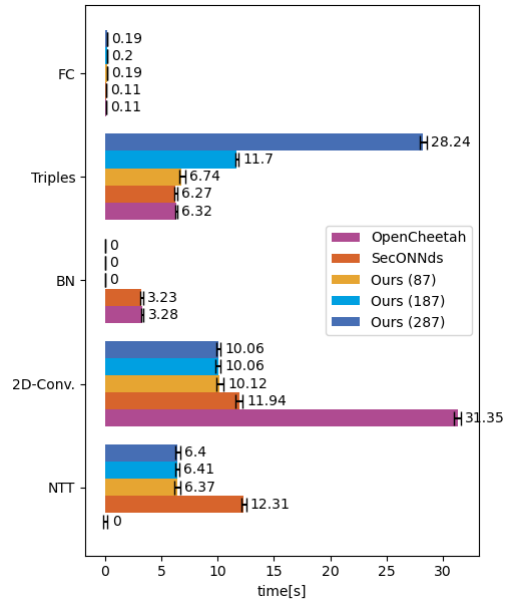


Figure 2: Comparing the runtime of different operations.

end-to-end time. The best overall time is achieved by using the ripple-carry adder approach. Here, only about 14.89% is done in the online phase.

Figure 2 illustrates how the operations perform across different frameworks.

We see that SecONNds and OpenCheetah still require some seconds for batch normalization. For fully connected layers, our implementation is slower by a couple of milliseconds, because we have to perform several conversions between 32-bit and 64-bit integers and also re-establish the connection between the two parties. Our three approaches primarily differ in the number of Boolean triples required and therefore, the time required to generate them. Finally, when it comes to convolution, we save about 6 s for NTT and only about 1 s for the computation. This seems strange as we have not modified the NTT-conversation at all. We suspect this may be due to measurement inaccuracies, as by default we do not distinguish between the NTT-phase and the computation phase. To investigate this, we increased the batch size to see how long convolutions take when there is no need for encoding the filters. With batch size set to two convolutions we measured about 6.82 s for NTT and about 23.35 s for the computation and with batch size set to ten

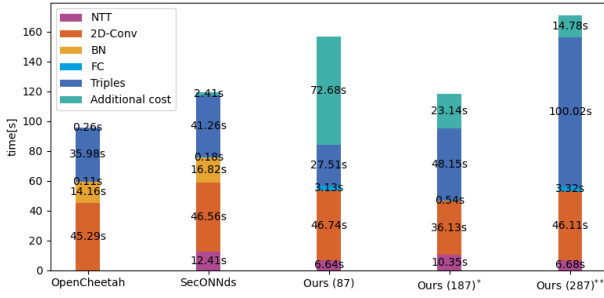


Figure 3: Evaluation of each operation and end-to-end times in the WAN setting (*uses separate send/receive threads; **uses online-phase optimizations).

we still got about 6.5 s for NTT and 128.83 s assuming that the first batch took about 10 s any subsequent batch takes about 13 s.

Finally, when making use of SIMD (vectors of 8 32 bit numbers) and parallelization via 32 processes, the last process takes about 1863.49 s, thus for an overall batchsize of 256 (8 (because of SIMD) · 32 (number of processes) = 256), each inference takes about $\frac{1863.49 \text{ s}}{256} = 7.28 \text{ s}$. This shows that utilizing parallelization improves the end-to-end times.

5.3. Wide Area Network

For these experiments, we set the network to a bandwidth of 200 Mbits⁻¹ and a latency of 20 ms.

Fig. 3 shows the runtime for each operation for ResNet50. With "Additional Cost", we refer to the time required for additional operations beyond the operations. This way, we can properly compare the end-to-end times. In our implementations, this also includes the online phase. We observe that our implementation with the ripple-carry adder, although it takes only approximately 90 s in its offline phase, requires about the same time in the online phase, diminishing the benefit. We notice that simple operations, such as FC, take slightly longer because we have to reestablish a connection between 32 threads, which in itself takes about 2 s but for function 187 we used a updated version where connections are established in parallel and reduced the only the connections required as for example FC only requires one port.

To (287) we applied some online phase optimizations outside of the scope of this paper. This can also be applied to (187) for only a minor performance loss in the offline phase of about 1.62 s but reduces the online-phase to about 21.49 s).

Finally, our results for the optimized version as introduced in Section 4.7, reduce the total time required for convolutions to about 46 s where the first ≈ 10 s are used for NTT. In total, we arrive at ≈ 102 s in the offline phase (including the key exchange ≈ 5.91 s). This greatly improves upon the work of SecONNds who require approx. 58.97 s and is about even with OpenCheetah (see Fig. 3). However, since NTT is only required for the first batch, our implementation has the edge when the main bottleneck is network speed. Nonetheless, this has no impact on our online phase, which adds an additional 23.14 s (using the parallel-prefix-adder). Note that for our function 187 in

Fig. 3 we used this version, whereas for 87 and 287 we used to sequential unoptimized implementation.

6. Conclusion and future work

We showed that using HE can be beneficial for more complex operations, and that most CPU-intensive operations can be performed in the offline phase using beaver triples. This way, the online phase accounts for only about 14.89 % of the total computation.

However, common approaches still need to be optimized, as the ciphertext quickly grows with the number of operations and is strongly dependent on the packing. Therefore, when creating an algorithm, one has to avoid sparse packing. Otherwise, the communication increases, but using less sparse encoding might require more operations, such as error correction or multiplications. Also, secret-sharing approaches become more efficient, as schemes like COT greatly reduce the required communication [12]. However, projects such as BumbleBee show that packing can still be optimized [7].

List of Acronyms

2PC	Secure Two-Party Computation
BN	Batch Normalization
COT	Correlated OT
CPU	Central Processing Unit
FC	Fully Connected Layer
GPU	Graphics Processing Unit
HE	Homomorphic Encryption
HPMPC	High-Throughput MPC
LAN	Local Area Network
MPC	Secure Multi-Party Computation
NTT	Number Theoretic Transform
OT	Oblivious Transfer
ReLU	Rectified Linear Unit
RLWE	Ring Learning with Errors
ROT	Random OT
SEAL	Simple Encrypted Arithmetic Library
SecONNds	Secure Outsourced Neural Network Interference
SIMD	Single Instruction, Multiple Data
WAN	Wide Area Network

References

- [1] 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 21)*. USENIX Association, Aug. 2021, pp. 2165–2182. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity21/presentation/patra>
- [2] C. Fedele, K. Butler, C. Petersen, and T. Lovelly, *Protecting Satellite Proximity Operations via Secure Multi-Party Computation*, 2024. [Online]. Available: <https://arc.aiaa.org/doi/abs/10.2514/6.2024-0271>
- [3] Y. Son, K. Han, Y. S. Lee, J. Yu, Y.-H. Im, and S.-Y. Shin, "Privacy-preserving breast cancer recurrence prediction based on homomorphic encryption and secure two party computation," *PLoS ONE*, vol. 16, no. 12, p. e0260681, 2021.

- [4] Z. Huang, W. jie Lu, C. Hong, and J. Ding, "Cheetah: Lean and fast secure Two-Party deep neural network inference," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 809–826. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity22/presentation/huang-zhicong>
- [5] C. Harth-Kitzerow, A. Suresh, Y. Wang, H. Yalame, G. Carle, and M. Annavaram, "High-throughput secure multiparty computation with an honest majority in various network settings," *Proceedings on Privacy Enhancing Technologies*, vol. 2025, no. 1, pp. 250–272, 2025.
- [6] S. Balla, "SecONNs: Secure outsourced neural network inference on imagenet," 2025. [Online]. Available: <https://arxiv.org/abs/2506.11586>
- [7] W. Lu, Z. Huang, Z. Gu, J. Li, J. Liu, C. Hong, K. Ren, T. Wei, and W. Chen, "BumbleBee: Secure two-party inference framework for large transformers," in *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. [Online]. Available: <https://www.ndss-symposium.org/ndss-paper/bumblebee-secure-two-party-inference-framework-for-large-transformers/>
- [8] J. Ma, Y. Zheng, J. Feng, D. Zhao, H. Wu, W. Fang, J. Tan, C. Yu, B. Zhang, and L. Wang, "SecretFlow-SPU: A performant and User-Friendly framework for Privacy-Preserving machine learning," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 17–33. [Online]. Available: <https://www.usenix.org/conference/atc23/presentation/ma>
- [9] X. Liu, Z. Liu, Q. L. 0002, K. X. 0002, and M. Xu, "Pencil: Private and extensible collaborative learning without the non-colluding assumption," in *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024. [Online]. Available: <https://www.ndss-symposium.org/ndss-paper/pencil-private-and-extensible-collaborative-learning-without-the-non-colluding-assumption/>
- [10] E. Barker, "Recommendation for key management: Part 1 - general," National Institute of Standards and Technology (NIST), Tech. Rep. SP 800-57 Part 1 Rev. 5, May 2020. [Online]. Available: <https://doi.org/10.6028/NIST.SP.800-57pt1r5>
- [11] "Microsoft SEAL (release 4.1)," <https://github.com/Microsoft/SEAL>, Jan. 2023, microsoft Research, Redmond, WA.
- [12] K. Yang, C. Weng, X. Lan, J. Zhang, and X. Wang, "Ferret: Fast extension for correlated ot with small communication," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1607–1626. [Online]. Available: <https://doi.org/10.1145/3372297.3417276>

Appendix

A.1 Our protocols

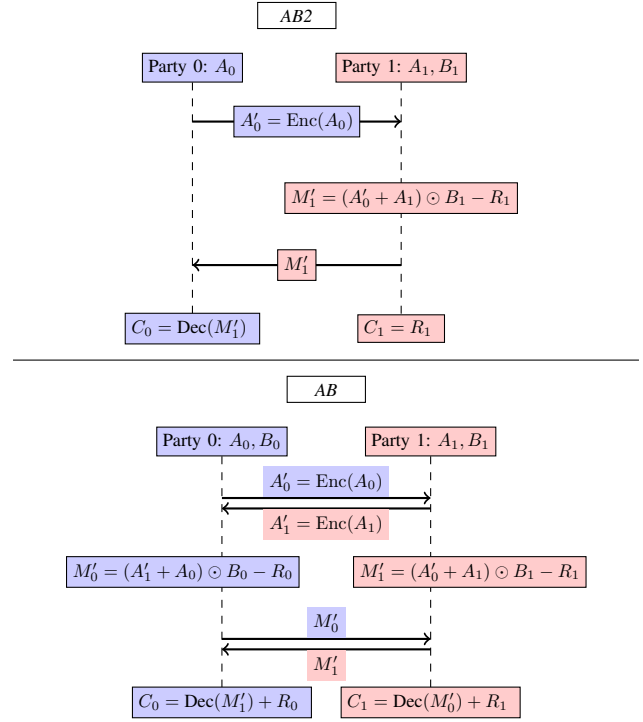


Figure 4: Protocols for beaver triple generation

Our protocols for beaver triple generation are depicted in Fig. 4. Red represents all operations done by party P_1 and blue, the operations by party P_0 has to do and all encrypted values are denoted with a single quote. Note, that AB generates the standard beaver triples, whereas $AB2$ assumes only one party has the filters. The resulting triple A_p, B_p, C_p for $p \in \{0, 1\}$ fulfill Equation (2).

$$(C_0 + C_1) = (A_0 + A_1) \odot (B_0 + B_1) \quad (2)$$