TECHNISCHE
UNIVERSITÄT
DARMSTADT

Master Thesis

Securely Realizing Output Privacy in MPC

Liang Zhao

July 8, 2022

ENCRYPTO
CRYPTOGRAPHY AND
PRIVACY ENGINEERING

Cryptography and Privacy Engineering Group
Department of Computer Science
Technische Universität Darmstadt

Supervisors: M.Sc. Helen Möllering
M.Sc. Oleksandr Tkachenko
Prof. Dr.-Ing. Thomas Schneider

Erklärung zur Abschlussarbeit gemäß §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Liang Zhao, die vorliegende Master Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Thesis Statement pursuant to §23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Liang Zhao, have written the submitted Master Thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

In the submitted thesis the written copies and the electronic version for archiving are identical in content.

Darmstadt, July 8, 2022

Liang Zhao

Abstract

Nowadays, the world has become an information-driven society where the distribution and processing of information is one important economic activity. However, the centralized database may contain sensitive data that would lead to privacy violations if the data or its aggregate statistics are disclosed. Secure Multi-Party Computation (SMPC) enables multiple parties to compute an arbitrary function on their private inputs and reveals no information beyond the computation result. Differential Privacy (DP) is a technique that can preserve the individual's privacy by perturbing the aggregate statistics with random noise. The hybrid approach combining SMPC and DP would provide a robust privacy guarantee and maintain the utility of the aggregate statistics. The theoretical definition of DP assumes precise noise sampling and arithmetic operations under real numbers. However, in the practical implementation of perturbation mechanisms, fixed-point or floating-point numbers are used to represent real numbers that lead to the violation of DP, as Mironov [Mir12] and Jin et al. [JMRO22] showed. This thesis explores the possibilities of *securely* generating distributed random noise in SMPC settings and builds a variety of perturbation mechanisms. Specifically, we evaluate the performance of fixed-point and floating-point arithmetic for noise generation in SMPC and choose the most efficient SMPC protocols to build the perturbation mechanisms.

Contents

1 Evaluation

Recall that the two challenges that we proposed in **??**:

1. What are the most efficient MPC protocols for arithmetic operations?
2. What are the most efficient sampling algorithms for differentially private mechanisms in MPC?

To answer the above questions, we implement and measure the performance of all the protocols we have discussed in **??** in a semi-honest scenario. First, we benchmark the performance of integer, fixed-point and floating-point operations in different MPC protocols (BGMW, AGMW, and BMR). Next, we choose the most efficient MPC protocols for arithmetic operations and implement the differentially private mechanisms and corresponding sampling algorithms. The code can be found at [Zha22];

Experimental Setup. The experiments are performed in five connected servers (Intel Core i9-7960X process, 128GB RAM, 10 Gbps network). We define two network settings to analyze the performance of our MPC-DP protocols.

1. LAN10: 10Gbit/s Bandwidth, 1ms RTT.
2. WAN: 100Mbit/s Bandwidth, 100ms RTT.

1.1 Arithmetic Operations Performance Evaluation

In this section, we provide extensive benchmarks for arithmetic protocols (cf. **??**). We choose the most efficient MPC protocols for arithmetic operations based on the benchmark result.

Fixed-Point Benchmarks TODO: add benchmark result of BMR protocols, BMR protocol is only 1x-1.5x faster than BGMW in division and slower in all other operations. Therefore, we choose BGMW as main protocols

Tab. 1.1 shows that division is the slowest operation for BGMW fixed-point arithmetic. The reason is that the circuit we generated using HyCC [BDK+18] have a deeper depth of 6 432. In the LAN setting, the 5-party division is 2.2× slower than the 3-party division, whereas, in the WAN setting, the factor was 1.1.

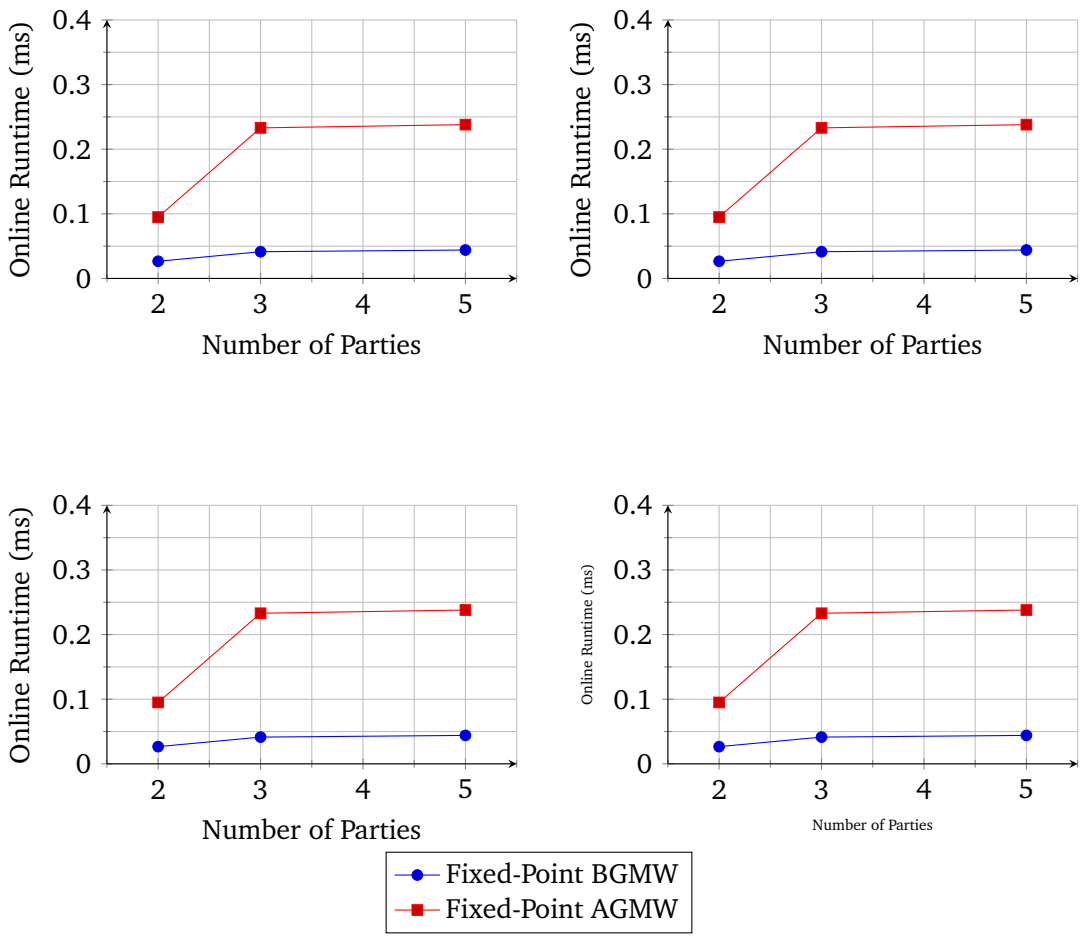

Figure 1.1: Performance Profile `member` types using the bdd-graph algorithm and rte algorithm, with different bdd cache allocation strategies

In Tab. 1.2, natural exponentiation has the longest online run-times. In the LAN setting, as the number of parties increases from 3 to 5, the online run-times of operation subtraction and square root increase at most by a factor 15× and 3.8×.

Comparing the online run-times of fixed-point arithmetic from Tab. 1.1 and Tab. 1.2, we can see that the BGMW-based fixed-point arithmetic (with $SIMD = 1$) is more efficient than the AGMW-based fixed-point arithmetic only in operations such as exp2, exp and conversion operation with integer, but slower in all other operations. However, when we apply the SIMD technique, BGMW-based fixed-point arithmetic (with $SIMD = 1000$) is $4 \times -4000\times$ faster than the AGMW-based fixed-point arithmetic for all the operations in the LAN setting. In the WAN setting, the BGMW-based fixed-point arithmetic is $35 \times -2800\times$ faster than the AGMW-based fixed-point arithmetic except for addition and multiplication operations. That means if the functionalities (e.g., *FOR* loop in $\Pi^{prot:TwoSideGeometric}$) can be parallelized into independent and identical operations, the BGMW-based fixed-point arithmetic is a better

Table 1.1: Online run-times in milliseconds (ms) of fixed-point (k=64, f=16) operations for the GMW (B) and TODO: add BMR protocol (Y). For the entries with $SIMD = 1\,000$, we specify the run-time of a single operation amortized over 1 000 SIMD values. We take the average over 10 protocol runs in the LAN and WAN environments.

		LAN			WAN		
Parties N	SIMD	N=2	N=3	N=5	N=2	N=3	N=5
FX_Add^B	1	27.49	29.22	39.75	507.80	516.25	556.10
FX_Sub^B	1	23.74	60.90	39.36	527.04	565.47	562.13
FX_Mul^B	1	197.14	199.17	219.78	1 487.11	1 772.54	1 829.94
FX_Div^B	1	25 690.22	26 924.64	59 848.43	336 749.59	354 113.67	391 287.07
FX_Lt^B	1	28.63	96.86	183.17	885.07	867.33	995.09
FX_Exp2^B	1	239.48	259.27	255.51	4 924.61	5 330.73	5 566.90
FX_Log2^B	1	2 880.34	2 898.00	2 887.24	17 599.92	19 747.61	22 030.32
FX_Exp^B	1	297.63	283.95	256.11	5 482.55	5 971.34	6 326.46
FX_Ln^B	1	2 910.26	2 995.85	2 985.79	18 903.29	19 957.76	22 555.63
FX_Sqrt^B	1	1 277.98	1 749.13	2 022.90	19 957.41	20 955.41	23 665.47
$Fx2Int^B$	1	14.35	15.53	17.53	675.69	795.63	948.14
$Fx2FL^B$	1	683.13	722.25	1 271.52	10 781.96	11 794.16	12 818.66
FX_Add^B	1 000	0.03	0.04	0.04	0.54	0.48	0.79
FX_Sub^B	1 000	0.01	0.01	0.04	0.51	0.56	0.52
FX_Mul^B	1 000	0.10	0.13	0.17	1.12	2.47	2.24
FX_Div^B	1 000	24.71	25.25	58.90	337.73	357.79	391.94
FX_Lt^B	1 000	0.05	0.15	0.14	0.79	0.80	1.00
FX_Exp2^B	1 000	0.21	0.41	0.47	4.75	5.68	6.14
FX_Log2^B	1 000	2.61	1.80	1.57	16.35	20.13	26.91
FX_Exp^B	1 000	0.24	0.45	0.55	5.66	6.37	6.65
FX_Ln^B	1 000	4.35	3.93	1.70	17.50	20.41	33.14
FX_Sqrt^B	1 000	4.62	4.33	5.11	19.44	21.60	32.91
$Fx2Int^B$	1 000	0.01	0.02	0.05	0.62	0.56	0.80
$Fx2FL^B$	1 000	0.25	0.40	0.55	7.36	9.64	10.38

option than AGMW-based fixed-point arithmetic in the LAN setting. Therefore, we choose BGMW-based fixed-point arithmetic to implement the MPC protocols.

Table 1.2: Online run-times in milliseconds (ms) of fixed-point (k=41, f=20) operations for the GMW (A). We take the average over 10 protocol runs in the LAN and WAN environments.

	LAN			WAN		
Parties N	N=2	N=3	N=5	N=2	N=3	N=5
FX_Add^A	0.10	0.23	0.24	0.19	0.25	0.28
FX_Sub^A	0.14	0.22	3.45	0.27	0.24	0.46
FX_Mul^A	16.05	38.24	25.20	531.76	437.77	448.93
FX_Div^A	219.16	755.79	1 650.56	8 968.31	10 198.43	11 210.29
FX_Lt^A	19.32	13.00	18.26	350.21	334.68	372.35
FX_Exp2^A	387.46	752.12	1 891.61	13 458.62	14 164.27	16 461.11
FX_Log2^A	181.10	215.64	292.53	4 371.94	4 716.72	5 565.20
FX_Exp^A	444.02	1 043.37	1 924.93	13 931.69	14 527.76	16 205.35
FX_Ln^A	223.29	280.02	308.78	4 864.74	5 049.57	5 652.10
FX_Sqrt^A	343.14	438.70	1 682.41	10 792.22	11 573.25	13 250.91
FX_Fx2FL^A	40.91	50.22	187.62	1 653.18	1 733.67	1 866.73
FX_Fx2Int^A	13.81	17.26	26.25	336.89	338.50	341.04

Floating-Point Benchmarks In Tab. 1.3, natural logarithm takes the longest online run-times in both LAN and WAN settings. As the number of parties grows from 3 to 5, the run-times of conversion operations to integer and fixed-point increase at most by a factor 3.08× and 3.66× ($SIMD = 1$), factor 2.2× and 1.9× ($SIMD = 1000$). Comparing the online run-times of floating-point arithmetic from Tab. 1.3 and Tab. 1.4, we can see that without applying SIMD, the BGMW-based floating-point arithmetic is faster ($1.3\times - 3\times$) than AGMW-based floating-point arithmetic in operations such as addition, subtraction, $<$, log2 and square root, but slower in all other operations. After applying the SIMD technique, the BGMW-based floating-point arithmetic is $50\times - 10911\times$ faster than AGMW-based floating-point arithmetic for all the operations in the LAN setting. Therefore, we choose BGMW-based floating-point arithmetic to implement MPC protocols for sampling algorithms.

Table 1.3: Online run-times in milliseconds (ms) of floating-point operations for the GMW (B) TODO: and BMR protocol (Y). For the entries with $SIMD = 1\,000$, we specify the run-time of a single operation amortized over 1 000 SIMD values. We take the average over 10 protocol runs in the LAN and WAN environments.

		LAN			WAN		
Parties N	SIMD	N=2	N=3	N=5	N=2	N=3	N=5
FL_Add^B	1	158.90	175.47	266.47	4 976.43	5 665.71	5 780.94
FL_Sub^B	1	151.38	167.19	252.31	5 053.17	5 599.27	5 970.80
FL_Mul^B	1	166.46	178.80	222.60	6 759.47	7 278.22	7 671.80
FL_Div^B	1	1 418.62	1 659.29	1 882.86	74 821.42	89 112.86	100 660.43
FL_Lt^B	1	47.08	52.42	132.53	1 638.13	1 751.99	1 840.56
FL_Exp2^B	1	1 958.28	2 391.79	3 392.22	112 465.43	129 661.24	140 737.75
FL_Log2^B	1	1 271.88	1 599.22	2 581.09	49 886.75	58 236.84	64 266.20
FL_Exp^B	1	2 122.94	2 564.97	3 461.87	118 774.84	137 115.63	150 439.77
FL_Ln^B	1	3 725.02	4 765.81	8 773.41	151 622.29	177 746.42	200 747.65
FL_Sqrt^B	1	887.72	1 081.30	1 514.39	44 584.20	51 846.96	57 450.22
$FL2Int^B$	1	327.69	451.41	1 390.65	12 943.05	14 093.49	16 199.04
$FL2Fx^B$	1	453.89	481.33	1 763.47	19 657.52	21 243.10	24 070.59
FL_Add^B	1000	0.23	0.34	0.61	4.73	5.18	5.81
FL_Sub^B	1000	0.23	0.31	0.55	4.85	5.37	5.47
FL_Mul^B	1000	0.23	0.29	0.27	6.67	7.30	7.85
FL_Div^B	1000	2.45	2.94	3.82	63.26	70.05	74.88
FL_Lt^B	1000	0.05	0.06	0.17	1.42	1.53	1.53
FL_Exp2^B	1000	1.78	2.05	2.66	84.15	96.90	105.37
FL_Log2^B	1000	1.19	1.59	2.23	42.75	48.91	52.82
FL_Exp^B	1000	1.94	2.28	2.57	90.90	104.04	113.61
FL_Ln^B	1000	4.12	5.57	7.54	128.44	146.59	163.34
FL_Sqrt^B	1000	0.14	0.30	0.59	2.48	2.89	3.30
$FL2Int^B$	1000	0.21	0.32	0.74	10.26	11.36	12.31
$FL2Fx^B$	1000	0.33	0.53	1.03	16.64	18.06	19.56

Table 1.4: Online run-times in milliseconds (ms) of floating-point operations for the GMW (A). We take the average over 10 protocol runs in the LAN and WAN environments. TODO: implement FL-FL2Fx and FL-FL2Int

	LAN			WAN		
Parties N	N=2	N=3	N=5	N=2	N=3	N=5
FL_Add^A	216.99	253.25	474.08	4 877.91	5 440.09	6 067.92
FL_Sub^A	216.63	314.82	491.58	5 000.17	5 356.96	6 113.30
FL_Mul^A	53.35	75.07	84.27	1 248.12	1 263.70	1 553.22
FL_Div^A	123.93	168.83	301.37	3 173.60	3 344.13	3 485.83
FL_Lt^A	66.03	84.35	197.75	1 361.24	1 566.96	1 779.53
FL_Exp2^A	569.54	953.94	1 784.02	9 105.86	10 054.34	13 147.79
FL_Log2^A	3 871.35	5 395.07	8 623.15	90 173.12	99 853.84	107 905.28
FL_Exp^A	620.11	966.76	1 684.20	9 748.59	10 695.65	11 560.34
FL_Ln^A	3 903.48	5 588.28	8 516.19	90 034.64	99 444.66	108 395.14
FL_Sqrt^A	1 575.21	2 372.36	3 603.36	31 263.47	33 778.76	36 776.32
FL_FL2Fx^A						
FL_FL2Int^A						

1.2 Evaluation of MPC Protocols for Differentially Private Mechanisms

Tab. 1.5 presents the combinations of different sampling algorithms to generate discrete Laplace random variables and discrete Gaussian random variables as discussed in **??**. We implement corresponding protocols in BGMW-based fixed/floating-point arithmetic. The benchmark results are listed in Tab. 1.6. We found that memory overflow ($>$ 128 GB) happends for $DLap_1^B$, $DGau_1^B$ and $DGau_2^B$ when we set the failure probability $p < 2^{-40}$. The reason is that for failure probability $p < 2^{-40}$, discrete random variables need a larger number of iterations. TODO: Therefore, we need to reimplement the MPC protocols for $DLap_1$, $DGau_1$ and $DGau_2$ with AGMW-based fixed/floating protocols.

Table 1.5: Overview of sampling algorithms for generating discrete Laplace/Gaussian random variables.

	Sampling Algorithms		
Random Variable	$Algo^{TwoSideGeo}$ [Tea20a]	$Algo^{DLap}$ [CKS20]	$Algo^{DGau}$ [CKS20]
$DLap_1$	•		
$DLap_2$		•	
$DGau_1$	•		•
$DGau_2$		•	•

Table 1.6: Online run-times in milliseconds (ms) of fixed/floating-point Laplace/Gaussian sampling protocols for the GMW (B). We take the average over 10 protocol runs in the LAN and WAN environments.

	LAN			WAN		
Parties N	N=2	N=3	N=5	N=2	N=3	N=5
$DLap_1^{B,FX}$	—	—	—	—	—	—
$DLap_1^{B,FL}$	—	—	—	—	—	—
$DLap_2^{B,FX}$	60 946.33	62 613.41	144 253.44	893 460.44	947 818.24	1 040 754.52
$DLap_2^{B,FL}$	48 874.17	49 236.48	105 650.76	730 002.77	782 549.19	861 691.71
$DGau_1^{B,FX}$	—	—	—	—	—	—
$DGau_1^{B,FL}$	—	—	—	—	—	—
$DGau_2^{B,FX}$	—	—	—	—	—	—
$DGau_2^{B,FL}$	—	—	—	—	—	—

We present the performance evaluation of the MPC protocols for differentially private mechanisms in Tab. 1.7.

Laplace Mechanisms. Recall that Integer-Scaling Laplace mechanism M_{ISLap} achieves differential privacy protection effect as Laplace mechanism textitsecurely by re-scaling a discrete Laplace random variable. In our work, we generate the discrete Laplace random variable with sampling protocol Π^{DLap} (cf. **??**) and set the failure probability as $p < 2^{-40}$. We can see that the M_{ISLap} with fixed-point implementation of Π^{DLap} is $1.09\times - 1.29\times$ slower than that with floating-point implementation. The reason is that the division operation in fixed-point is more expensive than floating-point for BGMW protocols.

For comparison, we implement the *insecure* Laplace mechanisms M_{Lap} [EKM+14]. Note that Mironov [Mir12] showed that M_{Lap} [EKM+14] suffered from floating-point attacks and proposed the snapping mechanism M_{SM} as a solution. The snapping mechanism M_{SM} has the best online run-times performance and it is $5380\times - 11940\times$ faster than M_{ISLap} and $1.32\times - 1.62\times$ faster than M_{Lap} [EKM+14]. However, it is worth to mention that the snapping mechanism M_{SM} introduces additional errors (beyond the necessary amount Laplace noise), that leads to a significant reduction in utility [Cov19; Tea20b]. In contrast, we could reduce the errors introduced by the Integer-Scaling Laplace mechanism M_{ISLap} by setting appropriate resolution parameter r (cf. **??**).

Gaussian Mechanisms. Recall that the Integer-Scaling Gaussian mechanism M_{ISLap} deploys the sampling protocol $\Pi^{SymmBinomial}$ under floating-point arithmetic to simulate the continuous Gaussian random variable. We implement $\Pi^{SymmBinomial}$ with BGMW-based floating-point arithmetic. We can see that the Integer-Scaling Gaussian mechanism M_{ISGau} is $2.08\times - 3.06\times$ faster than the Integer-Scaling Laplace mechanism M_{ISLap}.

Discrete Laplace Mechanisms. The discrete Laplace mechanism M_{DLap} deploys the same sampling protocol Π^{DLap} (cf. **??**) as M_{ISLap}. We also implement M_{DLap} [EKM+14] for comparison. It can be seen, that M_{DLap} [EKM+14] is at least $1068\times$ faster than our implementation of M_{DLap} in BGMW-based fixed/floating-point arithmetic in the LAN setting. However, the security of M_{DLap} [EKM+14] remains to prove as it applies similar noise generation procedure as M_{Lap} [EKM+14].

Discrete Gaussian Mechanisms. TODO: We encounter memory overflow when implement BGMW-based M_{DGau}, try to implement it in AGMW-based floating-point arithmetic

Table 1.7: Online run-times in milliseconds (ms) for differentially private mechanisms for the GMW (B). We specify the run-time of a single operation amortized over corresponding SIMD values. We take the average over 10 protocol runs in the LAN and WAN environments.

			LAN			WAN		
Parties N	SIMD	Protocol	N=2	N=3	N=5	N=2	N=3	N=5
M_{Lap} [EKM+14] (insecure)	1000	B, FL	7.58	8.36	11.08	197.68	223.92	248.53
M_{SM} (**this work**)	1000	B, FL	4.68	5.77		145.70	168.74	182.60
M_{ISLap} (**this work**)	1	B, FX	72 296.53	71 352.89	160 443.40	949 907.44	994 913.88	1 091 406.72
M_{ISLap} (**this work**)	1	B, FL	55 858.07	47 186.90	111 232.66	846 435.93	908 322.81	1 001 687.22
M_{ISGau} (**this work**)	1	B, FL	18 916.06	19 413.08	26 929.39	394 248.57	435 126.28	470 813.95
M_{DLap} [EKM+14]	1000	B, FX	2.26	3.26	134.97	19.01	24.11	69.88
M_{DLap} [EKM+14]	1000	B, FL	5.91	6.60	9.06	144.93	163.96	177.91
M_{DLap} (**this work**)	1	B, FX	60 946.33	62 613.41	144 253.44	893 460.44	947 818.24	1 040 754.52
M_{DLap} (**this work**)	1	B, FL	48 874.17	49 236.48	105 650.76	730 002.77	782 549.19	861 691.71
M_{Dau} (**this work**)	1	B, Fx	—	—	—	—	—	—
M_{Dau} (**this work**)	1	B, FL	—	—	—	—	—	—

List of Figures

List of Tables

List of Abbreviations

SMPC Secure Multi-Party Computation

DP Differential Privacy

Bibliography

[BDK+18] N. BÜSCHER, D. DEMMLER, S. KATZENBEISSER, D. KRETZMER, T. SCHNEIDER. **"HyCC: Compilation of hybrid protocols for practical secure computation"**. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 847–861.

[CKS20] C. L. CANONNE, G. KAMATH, T. STEINKE. **"The discrete gaussian for differential privacy"**. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 15676–15688.

[Cov19] C. COVINGTON. **"Snapping Mechanism Notes"**. 2019.

[EKM+14] F. EIGNER, A. KATE, M. MAFFEI, F. PAMPALONI, I. PRYVALOV. **"Differentially private data aggregation with optimal utility"**. In: *Proceedings of the 30th Annual Computer Security Applications Conference*. 2014, pp. 316–325.

[JLL+19] K. JÄRVINEN, H. LEPPÄKOSKI, E.-S. LOHAN, P. RICHTER, T. SCHNEIDER, O. TKACHENKO, Z. YANG. **"PILOT: Practical privacy-preserving indoor localization using outsourcing"**. In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2019, pp. 448–463.

[JMRO22] J. JIN, E. MCMURTRY, B. RUBINSTEIN, O. OHRIMENKO. **"Are We There Yet? Timing and Floating-Point Attacks on Differential Privacy Systems"**. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society. 2022, pp. 1547–1547.

[Mir12] I. MIRONOV. **"On significance of the least significant bits for differential privacy"**. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. 2012, pp. 650–661.

[MRT20] P. MOHASSEL, M. ROSULEK, N. TRIEU. **"Practical Privacy-Preserving K-means Clustering"**. In: *Proceedings on Privacy Enhancing Technologies* 2020.4 (2020), pp. 414–433.

[Tea20a] G. D. P. TEAM. **"Google's differential privacy libraries"**. 2020.

[Tea20b] G. D. P. TEAM. **"Secure Noise Generation"**. 2020.

[Zha22] L. ZHAO. **"Securely Realizing Output Privacy in MPC"**. In: GitHub, 2022.

A Appendix

A.1 MPC Protocols

Oblivious Array Access $\Pi^{ObliviousSelection}$ is inspired by the works [JLL+19; MRT20]. $\Pi^{ObliviousSelection}$ uses the inverted binary tree, i.e., the leaves represent input elements, and the root represents the output element. The tree has $\log_2 \ell$-depth for input array of length ℓ and each node hold two shares: $\langle c \rangle^B$ and $\langle \boldsymbol{y} \rangle^B$. Fig. A.1 shows an example of the inverted binary tree. For $i \in [0, \ell - 1]$ and $j \in \left[\log_2 \ell\right]$, the values of node $(\left(\left\langle c_{a,b} \right\rangle, \left\langle \boldsymbol{y_{a,b}} \right\rangle^B\right))$ in the j-th layer are computed with the value of two nodes (with value $\left(\langle c_a \rangle, \langle \boldsymbol{y_a} \rangle^B\right)$ and $\left(\langle c_b \rangle, \langle \boldsymbol{y_b} \rangle^B\right)$) in the $j-1$-th layer as follows:

$$\left(\left\langle c_{a,b} \right\rangle, \left\langle \boldsymbol{y_{a,b}} \right\rangle^B\right) = \begin{cases} \left(\langle c_a \rangle, \langle \boldsymbol{y_a} \rangle^B\right), & \text{if } \langle c_a \rangle == 1 \\ \left(\langle c_b \rangle, \langle \boldsymbol{y_b} \rangle^B\right), & \text{if } \langle c_a \rangle == 0 \wedge \langle c_b \rangle == 1 \\ (0,0) & \text{if } \langle c_a \rangle == 0 \wedge \langle c_b \rangle == 0, \end{cases} \tag{A.1}$$

which is equivalent to

$$\left\langle c_{a,b} \right\rangle = \langle c_a \rangle \oplus \langle c_b \rangle \oplus (\langle c_a \rangle \wedge \langle c_b \rangle) \tag{A.2}$$

$$\begin{aligned} \left\langle \boldsymbol{y_{a,b}} \right\rangle = \quad & (\langle c_a \rangle \oplus \langle c_b \rangle) \cdot \left(\langle \boldsymbol{y_a} \rangle^{B,UINT} \cdot \langle c_a \rangle \oplus \langle \boldsymbol{y_b} \rangle^{B,UINT} \cdot \langle c_b \rangle\right) \\ & \oplus (\langle c_a \rangle \wedge \langle c_b \rangle) \cdot \langle \boldsymbol{y_a} \rangle^{B,UINT} \end{aligned} \tag{A.3}$$

The nodes is evaluated from the 1th layer until the root.

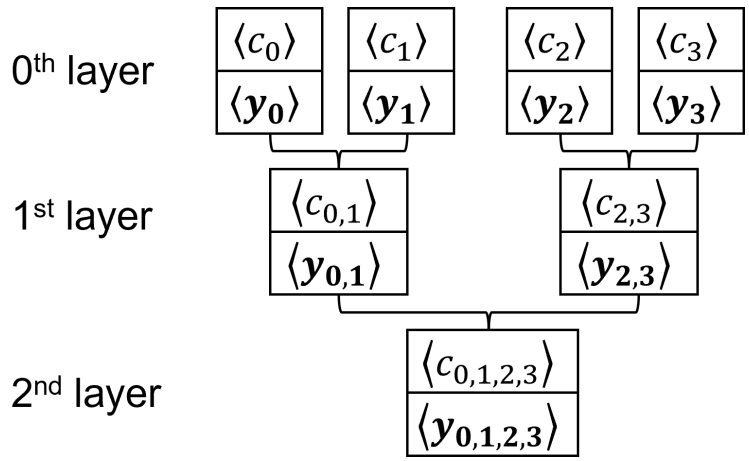

Figure A.1: Example inverted binary tree for $\Pi^{ObliviousSelection}$.