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 GitHub [1];

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 compare the performance of arithmetic operations in different data types (**BGMW!** (**BGMW!**) fixed/floating-point, **AGMW!** (**AGMW!**) fixed/floating-point). All the arithmetic operations and underlying protocols were tested for correctness with Google Test [2]. Specifically, the following arithmetic operations were benchmarked: addition, subtraction, multiplication, division, exp2, log2, square root, comparison (<), and data type conversions. We measure the total runtime (offline + online) in microseconds (ms) and average the runtime results over 10 iterations.

[1]https://github.com/liangzhao-darmstadt/MOTION
[2]https://github.com/google/googletest

Fixed-Point Benchmarks. In the experiments, we use the **BGMW!** fixed-point numbers (total bit-length: k=64, fraction bit-length: f=16). The **AGMW!** fixed-point numbers are represented with 128-bit signed integers and have fraction bit-length f=16. In Fig. 1.1, we compare the **BGMW!** and **AGMW!** fixed-point arithmetic operations. As the plot suggest, addition and subtraction are very fast for **AGMW!** fixed-point because they could be computed locally with precomputed Multiplication Triples (MTs) (cf. **??**). For the multiplication operation, the **BGMW!** and **AGMW!** fixed-point have very close performance, while the arithemtic truncation operation (cf. ??) is the major overhead of **AGMW!** fixed-point. The division operation is a expensive operation for both **BGMW!** and **AGMW!** fixed-point, where the **AGMW!** fixed-point is $1.21\times - 2.67\times$ faster than the **BGMW!** fixed-point. The base-2 exponential function (exp2) and the binary logarithm (log2) are substantially faster for the **BGMW!** than the **AGMW!** fixed-point (exp2: $7.64\times - 11.98\times$, log2: $75.84\times - 7.43\times$). For square root operation, the **BGMW!** fixed-point and **AGMW!** fixed-point have close performance in the Two-Party Computation (2PC) and Three-Party Computation (3PC) settings, whereas **BGMW!** fixed-point is 2.06× faster than the **AGMW!** fixed-point in the five-party settings. The comparision operation between **BGMW!** and **AGMW!** fixed-point are very close to each other, while the **BGMW!** fixed-point is more efficient as the number of computation parties increases. The conversion operation from fixed-point to integer is up to 22.93× faster for the **BGMW!** than the **AGMW!** fixed-point. In **BGMW!** fixed-point, the conversion to integer is can be realized with the arithmetic bit-shifting operation (cf.??) and integer comparison. However, the **AGMW!** fixed-point requires to apply expensive arithmetic right-shifting operations multiple times to truncate the fraction part and round it to the closest integer. The remaining conversion operation (fx2fl) is up to 2.10× faster for the **AGMW!** than the **BGMW!** fixed-point. Finally, we choose **BGMW!** fixed-point as the primary data type for fixed-point operations.

Floating-Point Benchmarks. We use the **BGMW!** floating point numbers that have 64 Boolean share bits. In contrast, the **AGMW!** floating-point numbers are represented by a quadruple (v, p, z, s) (cf.??), where v, p, z, and s are 128-bit arithmetic shares. As Fig. 1.2 shows, the **BGMW!** floating-point is faster than the **AGMW!** floating-point in almost all the arithmetic operations except for the division operation in 2PC settings. For addtion, subtraction and multiplication, the **BGMW!** floating-point is $1.59\times - 4.22\times$ faster than the **AGMW!** floating-point. The primary overhead of **AGMW!** is the truncation and bit-decomposion operations [ABZS12]. The division operation is slightly faster for the **BGMW!** than the **AGMW!** floating-point only in the five-party setting. For operation base-2 exponential function (exp2), binary logarithm (log2) and square root, the **BGMW!** floating-point is substantially faster than the **AGMW!** floating-point by a factor up to 13.94×. The reason for the low efficient performance of the **AGMW!** floating-point is the mulitply invocations of floating-point multiplication, where the floating-point multiplication operation consists of the expensive arithmetic right shifting operation and arithmetic share multiplication. The comparison operation is of **AGMW!** floating-point is more expensive than the **BGMW!** because of the arithmetic share comparision and arithmetic right shifting operations. For the remaining conversion operations (fl2int and fl2fx), the **BGMW!** floating-point is more efficient as the conversion to integer or fixed-point can be realized bit manipulation at low cost. Based on the above analysis, we choose **BGMW!** floating-point for floating-point arithemtic operations.

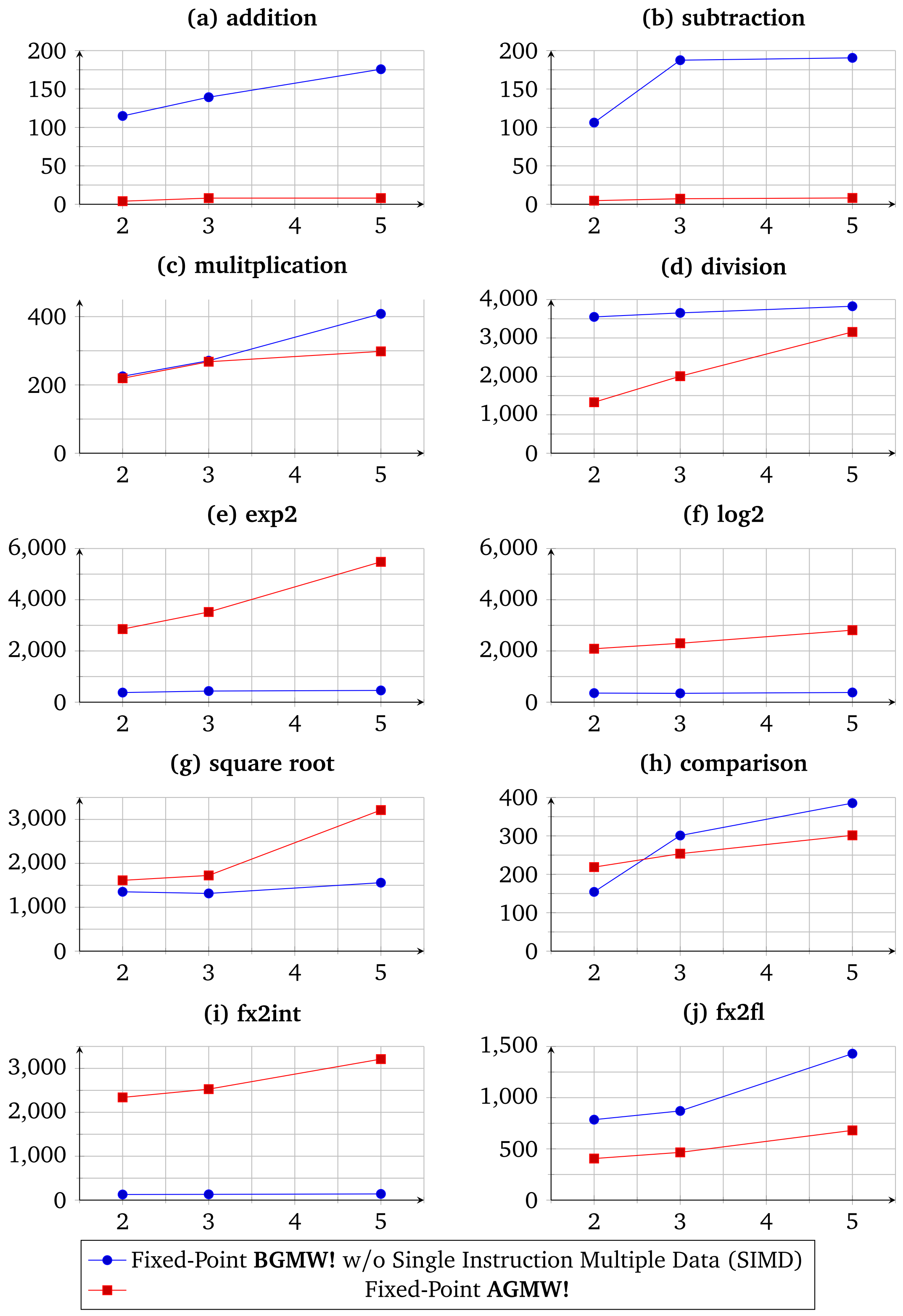

Figure 1.1: Total runtime in microseconds (y-axis) for fixed-point arithmetic operations for the **BGMW!** (B) [GMW87] and **AGMW!** protocol in WAN test encironments (cf. § 1) with $2, 3, 5$ parties (x-axis). We take the average results over 10 protocol runs.

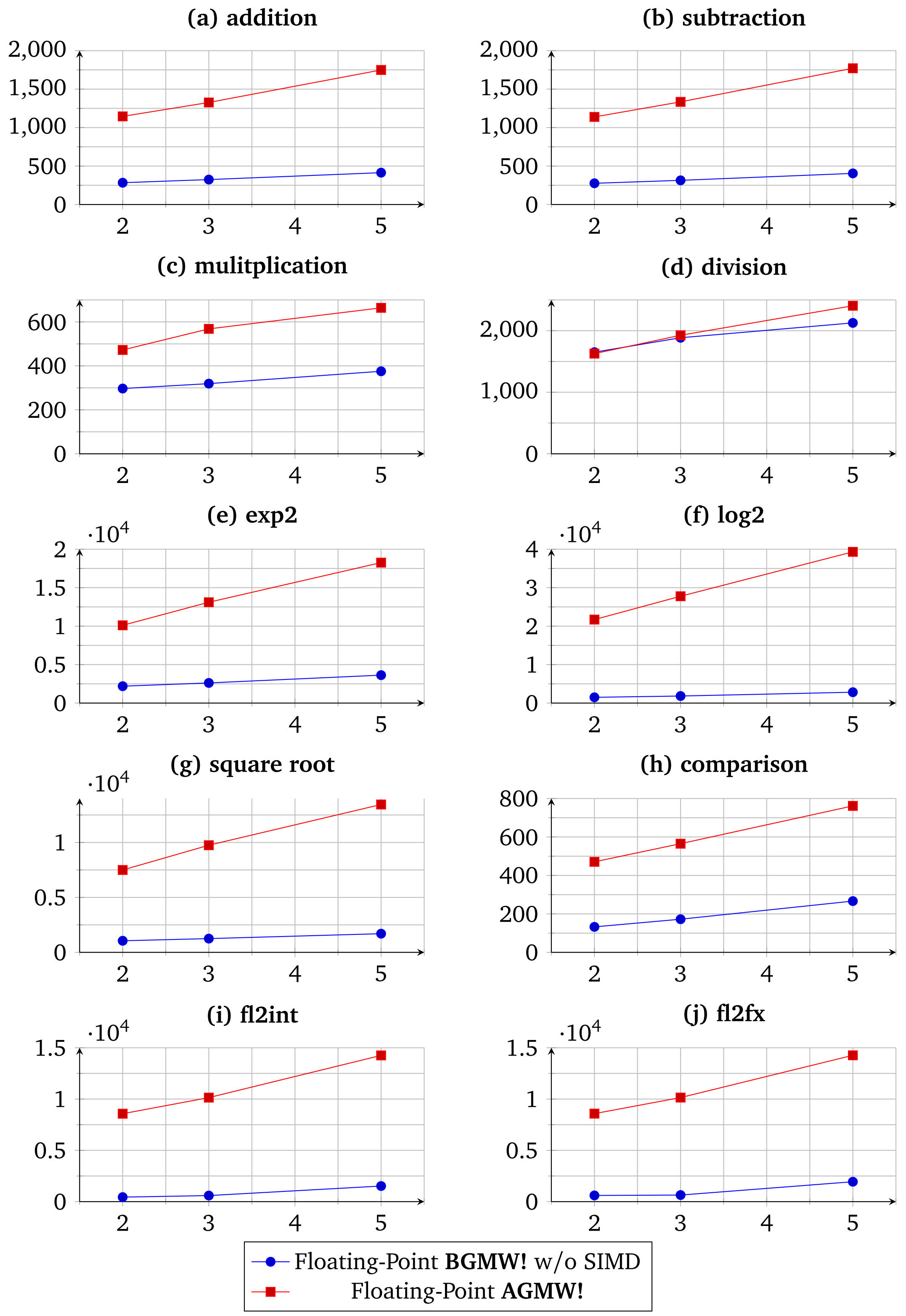

Figure 1.2: Total runtime in microseconds (y-axis) for floating-point arithmetic operations for the **BGMW!** (*B*) [GMW87] and **AGMW!** protocol in WAN test encironments (cf. § 1) with $2, 3, 5$ parties (x-axis). We take the average results over 10 protocol runs.

1.2 Differentially Pirvate Mechanism Benchmarks

We present the performance evaluation of the MPC protocols for differentially private mechanisms in Tab. 1.1.

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; Tea20]. 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 1.1: 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
MTs	Multiplication Triples
2PC	Two-Party Computation
3PC	Three-Party Computation
SIMD	Single Instruction Multiple Data

Bibliography

[ABZS12] M. Aliasgari, M. Blanton, Y. Zhang, A. Steele. **“Secure computation on floating point numbers”**. In: *Cryptology ePrint Archive* (2012).

[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.

[GMW87] O. Goldreich, S. Micali, A. Wigderson. **“How to play ANY mental game”**. In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 1987, pp. 218–229.

[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.

[Tea20] G. D. P. Team. **“Secure Noise Generation”**. 2020.

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.

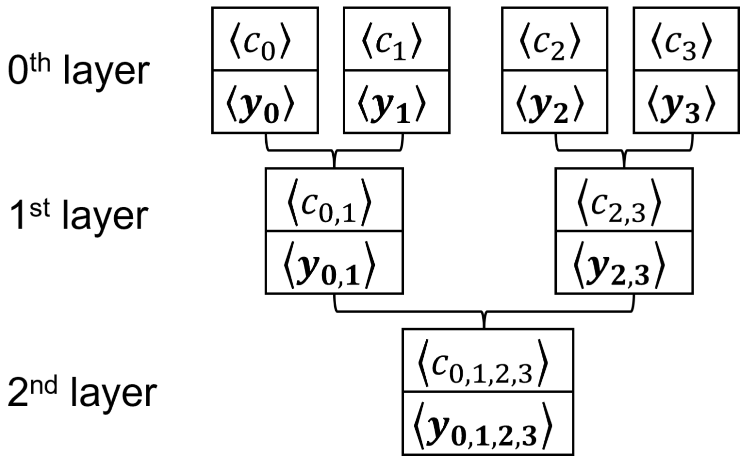
0th layer
1st layer
2nd layer
$\langle c_0 \rangle$
$\langle \boldsymbol{y_0} \rangle$
$\langle c_1 \rangle$
$\langle \boldsymbol{y_1} \rangle$
$\langle c_2 \rangle$
$\langle \boldsymbol{y_2} \rangle$
$\langle c_3 \rangle$
$\langle \boldsymbol{y_3} \rangle$
$\langle c_{0,1} \rangle$
$\langle \boldsymbol{y_{0,1}} \rangle$
$\langle c_{2,3} \rangle$
$\langle \boldsymbol{y_{2,3}} \rangle$
$\langle c_{0,1,2,3} \rangle$
$\langle \boldsymbol{y_{0,1,2,3}} \rangle$

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