UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Practical, Scalable, and Efficient Privacy-Preserving Computation

Permalink
https://escholarship.org/uc/item/9zq7751x

Author
Hussain, Siam Umar

Publication Date
2021

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Practical, Scalable, and Efficient Privacy-Preserving Computation

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Electrical Engineering (Computer Engineering)

by

Siam Umar Hussain

Committee in charge:

Professor Farinaz Koushanfar, Chair
Professor Andrew B. Kahng
Professor Ryan Kastner
Professor Bill Lin
Professor Tajana Simunic Rosing

2021

Copyright

Siam Umar Hussain, 2021

All rights reserved.

The Dissertation of Siam Umar Hussain is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2021

TABLE OF CONTENTS

Dissertation Approval Page iii

Table of Contents iv

List of Figures ix

List of Tables xii

Acknowledgements xiv

Vita xvii

Abstract of the Dissertation xix

Chapter 1 Introduction 1
1.1 Efficient and Scalable MPC Frameworks 3
1.2 General Purpose Hardware Platform for Privacy-Preserving Computation 5
1.3 Co-design and Optimization of Privacy-Preserving Computation and Hardware . 5
1.4 Real-World Applications 6
1.5 Co-optimization of Crypto Primitives and ML Inference 8

Chapter 2 Background 10
2.1 Notations 10
2.2 Secure Multi-Party Computation (MPC) 10
2.3 Oblivious Transfer 11
2.4 Yao's Garbled Circuit 12
2.4.1 Garbled Circuit Optimizations 13
2.4.2 Extension of GC for the Malicious Security Model 14
2.5 Beaver-Micali-Rogaway 15
2.6 Arithmetic Sharing 16
2.6.1 Addition and Multiplication in AS 17
2.7 Machine Learning Layers 17

Chapter 3 Efficient and Scalable MPC Frameworks 19
3.1 Overview 19
3.1.1 Automatic Generation of Optimized Boolean Logic 20
3.1.2 Rich Programming Paradigm 21
3.1.3 Scalability in Terms of Memory Footprint 22
3.1.4 Reliability 24
3.1.5 Evaluation Results 24
3.1.6 Summary of Contributions 25
3.2 Netlist Generation through HDL Synthesis 25
3.2.1 Synthesis Flow 26

3.2.2 Offline Circuit Synthesis . . . 27
3.2.3 Adaptation to BMR and GMW protocols . . . 28
3.3 Execution Flow of the GC Back-end . . . 29
3.3.1 Function Composition Formats . . . 29
3.3.2 Scalability Analysis . . . 30
3.4 Program Interface . . . 32
3.4.1 Protocol Instantiation . . . 32
3.4.2 Variables . . . 33
3.4.3 Functional Building Blocks . . . 33
3.4.4 Neural Network Building Blocks . . . 35
3.4.5 Cautions . . . 36
3.5 Evaluation of GC Frameworks . . . 37
3.5.1 Synthesis . . . 37
3.5.2 Runtime and Memory Footprint of Matrix-multiplication . . . 38
3.5.3 Runtime and Memory Footprint of CNN Inference with LeNet-5 . . . 40
3.5.4 Benchmarking the Program Interface . . . 41
3.6 Evaluation of BMR Framework . . . 41
3.6.1 Auction . . . 42
3.6.2 Voting . . . 43
3.7 Brief Overview of Existing GC Frameworks . . . 43
3.8 Summary . . . 47

Chapter 4 General Purpose Hardware Platform for Privacy-Preserving Computation . 48
4.1 Overview . . . 48
4.1.1 FPGA vs GPU as Acceleration Platform . . . 50
4.1.2 Summary of Contributions . . . 50
4.2 Global Flow . . . 51
4.2.1 Security Model and Terminology . . . 51
4.2.2 System Setup . . . 51
4.2.3 Client-Server Model . . . 52
4.2.4 Netlist Format . . . 53
4.2.5 Execution Steps of FASE . . . 54
4.3 Architecture of FASE . . . 55
4.3.1 Key Generator . . . 56
4.3.2 Garbling Engine . . . 56
4.3.3 Control Logic . . . 57
4.3.4 Memory Management . . . 58
4.3.5 Collector . . . 60
4.4 Scheduling the Gates . . . 61
4.4.1 Setting the priority . . . 62
4.4.2 Adding Gates to the Queue . . . 63
4.5 Evaluation . . . 63
4.5.1 Benchmark Functions . . . 63
4.5.2 Resource Utilization . . . 65

4.5.3 Evaluation of Scheduling and Memory Management . . . 66
4.5.4 Comparison with Previous Work . . . 66
4.5.5 Improvement in Throughput over Software Approach . . . 67
4.6 Summary . . . 69

Chapter 5 Custom Co-design and Optimization of Privacy-Preserving Computation and Hardware . . . 70
5.1 Overview . . . 70
5.1.1 Summary of Contributions . . . 72
5.2 Global Flow . . . 73
5.2.1 Security Model . . . 73
5.2.2 System Setup . . . 73
5.2.3 Client-Server Model . . . 74
5.3 Architecture of MAXelerator . . . 75
5.3.1 Segment 1: MUX_ADD . . . 76
5.3.2 Segment 2: TREE . . . 77
5.3.3 Accumulator and Support for Signed Inputs . . . 77
5.4 Hardware Setting and Results . . . 78
5.4.1 GC Engine . . . 78
5.4.2 Label Generator . . . 79
5.4.3 Resource Utilization . . . 80
5.4.4 Performance Comparison with the Prior-art GC Implementation . . . 81
5.5 Practical Design Experiments . . . 82
5.5.1 Deep Learning Benchmarks . . . 82
5.5.2 Generic ML Applications . . . 83
5.6 Summary . . . 85

Chapter 6 Real-World Applications . . . 86
6.1 Overview . . . 86
6.2 Secure Localization for Smart Cars . . . 87
6.2.1 Summary of Contributions . . . 90
6.2.2 Triangle Localization Algorithm . . . 90
6.2.3 Related Work . . . 92
6.2.4 Global Flow . . . 94
6.2.5 Protocol with Yao's GC . . . 95
6.2.6 Protocol with BMR . . . 100
6.2.7 Effect of the Motion of Cars . . . 101
6.2.8 Distance Compensation . . . 102
6.2.9 Netlist Generation . . . 103
6.2.10 Invocation of the MPC Protocols . . . 105
6.2.11 Evaluation: Error Analysis . . . 106
6.2.12 Evaluation: Circuit Synthesis . . . 108
6.2.13 Evaluation: Timing . . . 109
6.3 Authentication with Noisy Keys . . . 110

6.3.1 Summary of Contributions 113
6.3.2 Physical Unclonable Function (PUF) 113
6.3.3 Related Work 115
6.3.4 Threat Model 118
6.3.5 Authentication Function 119
6.3.6 Protocol Initialization 120
6.3.7 Protocol for Binary Response 121
6.3.8 Extension for Integer Response 122
6.3.9 Security of the Authentication Function 124
6.3.10 Security of the Authentication Protocol 128
6.3.11 Generating GC Netlist 130
6.3.12 Implementing LSH 131
6.3.13 Evaluation Settings 133
6.3.14 Evaluation of the Authentication Protocol 133
6.3.15 Evaluation of Protocol for Integer Response 134
6.4 Privacy Preserving k-Nearest Neighbor Search 135
6.4.1 Summary of Contributions 136
6.4.2 Related Work 137
6.4.3 Distance Function 137
6.4.4 Generation of Netlist 137
6.4.5 Combinational Garbled Circuit 138
6.4.6 Sequential Garbled Circuit 139
6.4.7 1-NNS in Multi-Party Setting 141
6.4.8 Evaluation: Memory Footprint of 1-NNS 142
6.4.9 Evaluation: Timing of 1-NNS 142
6.4.10 Evaluation: Memory Footprint of k-NNS 143
6.5 Private Set Intersection 144
6.5.1 Circuit Design 144
6.5.2 Evaluation 149
6.6 Summary 150

Chapter 7 Co-optimization of Crypto Primitives and ML Inference 151
7.1 Overview 151
7.2 Related Work 156
7.2.1 Cryptographic Optimization 156
7.2.2 ML Optimization 157
7.3 Global Flow and Threat Model 158
7.3.1 Threat Model 159
7.4 COINN Model Customization 160
7.4.1 Ciphertext-aware Quantization 160
7.4.2 Factored Matrix-Multiplication 161
7.4.3 Automated Parameter Configuration 163
7.5 Cryptographic Protocols 163
7.5.1 Matrix-Multiplication 164

7.5.2 Linear Layers in the Amortized Setting . . . 167
7.5.3 Non-linear Layers . . . 169
7.5.4 Cost Breakdown and Comparison with Previous Works . . . 170
7.6 Oblivious BNN Inference . . . 171
7.6.1 Binary Matrix Multiplication . . . 172
7.6.2 Nonlinear Layers . . . 173
7.6.3 Training Adaptive BNN . . . 173
7.7 Evaluation of COINN: Generic DNN Inference . . . 173
7.7.1 Evaluation of COINN Optimizations . . . 175
7.7.2 Comparison with Prior Work . . . 179
7.7.3 Model Customization Runtime . . . 181
7.7.4 Evaluation on Microbenchmarks . . . 182
7.8 Evaluations of BNN Inference . . . 183
7.8.1 Evaluating Flexible BNNs . . . 184
7.8.2 Oblivious Inference . . . 185
7.9 Summary . . . 188

Chapter 8 Conclusion and Open Challenges . . . 189

Appendix A Command Line Options and Available Functions in TinyGarble2 . . . 192

Appendix B Architecture of FASE . . . 195

Bibliography . . . 197

LIST OF FIGURES

Figure 3.1. Memory usage (MB) for matrix multiplication through GC. Batch size limitation is not applied to TinyGarble2. 39

Figure 3.2. Trade-off between the run-time and memory footprint for matrix multiplication through TinyGarble2 in the malicious setting. 39

Figure 4.1. FASE system architecture on the server side. 52

Figure 4.2. Architecture of FASE. (Please see Figure B.1 at Appendix B for an enlarged version.) 55

Figure 4.3. Wrapper module around the BRAM of Output Keys. 59

Figure 4.4. Different types of gate dependencies. 62

Figure 5.1. System configuration of MAXelerator framework. 74

Figure 5.2. Schematic of the tree-base multiplication. 76

Figure 5.3. The high-level configuration and functionality of the parallel GC cores in segment 1: MUX_ADD 77

Figure 5.4. Percentage resource utilization per MAC for different bit-widths. 80

Figure 6.1. Triangle Localization Algorithm. The lost car is Q and the assisting cars are A, B, and C. The calculated location of Q is the centroid of the triangle DEF. 91

Figure 6.2. Overview of the Localization Algorithm 94

Figure 6.3. The regions of uncertainty for car A in locating the other cars. The uncertainty region of the lost car Q is marked with stripes and the uncertainty region of the other two assisting cars B and C is marked with dots. 99

Figure 6.4. The $TriLoc$ netlist to compute the location of the lost car Q with help from three assisting cars A, B, and C through the BMR protocol. Only the netlist for computing the vertex D is shown in detail. 100

Figure 6.5. Illustration of parallel invocations of GC protocol. 105

Figure 6.6. Error Analysis. 107

Figure 6.7. Authentication protocol. 118

Figure 6.8. Extracting multiple LSH bits from single random permutation π. 132

Figure 6.9. Illustration of the actual path, Euclidean distance and taxicab distance 138

Figure 6.10. Combinational circuit for 1-NN. It consists of n taxicab distance and $(n-1)$ min modules. 139

Figure 6.11. Sequential circuit for 1-NNS. It consists of 1 taxicab distance and 1 min module. For a dataset of size n, the circuit is required to be garbled/evaluated n times. 140

Figure 6.12. Sequential circuit for k-NNS. It consists of 1 taxicab distance, k min, and $k-1$ max modules. It requires to be evaluated n times where n is the size of the dataset S. 141

Figure 6.13. Comparison of memory footprints of 1NNS with combinational and sequential approach 142

Figure 6.14. Comparison of garbling times of 1NNS with combinational and sequential approach 143

Figure 6.15. Memory footprint of k-NNS with sequential approach 143

Figure 6.16. High-level circuit description of the Sort-Merge-Compare-Shuffle for Private Set Intersection. Three operations are performed at each stage: merge, compare, and sort. 145

Figure 7.1. Accuracy and secure inference runtime of a 7-layer DNN on CIFAR-10 dataset using prior work: Delphi [1], SafeNet [2], XONN [3], AutoPrivacy [4], and CrypTFlow2 [5]. The ★ symbol represents COINN. 154

Figure 7.2. Overview of COINN. The plaintext model customization is only performed once per DNN and provides the optimized network for COINN secure inference. 158

Figure 7.3. Example 4×4 weight matrix approximated via clustering with $V = 4$. The approximated matrix W can be represented as a tuple $(C, \widetilde{W})$. 162

Figure 7.4. Plaintext operations and their equivalent ciphertext realization in COINN oblivious inference framework. 164

Figure 7.5. Effect of quantization bitwidth on communication cost (bars) and accuracy (curve). The numbers on the horizontal axis show the bitwidth for homogeneous quantization of weights/inputs across all layers. Q represents the heterogeneous bitwidths found by COINN. 176

Figure 7.6. Heterogeneous parameters across ResNet-32 layers found by COINN configurator. (a) Quantization bitwidths. (b) Number of clusters V. 177

Figure 7.7. Effect of factored multiplication on inference accuracy and communication cost of linear operations. Q represents the baseline quantized DNN. Numbers to its left represent homogeneous V for all layer weights. Q+C represents heterogeneous V configuration found by COINN. 178

Figure 7.8. Communication for baseline and COINN optimized models, where Q represents quantized model and Q+C further applies clustering to enable factored multiplication. 178

Figure 7.9. Breakdown of setup and amortized times for the under LAN and WAN settings. 179

Figure 7.10. CIFAR-10 test accuracy of each architecture at different widths. Our Adaptive BNN trains a single network that can operate at all widths, whereas previous work (XONN) trains a separate BNN per width 184

Figure 7.11. Runtime and communication cost of each architecture at different widths . 184

Figure 7.12. Improvements in LAN runtime and communication compared to XONN. Our protocols achieve 2× to 12× in runtime and 5× to 12× communication reduction 185

Figure 7.13. Breakdown of communication cost at linear and nonlinear layers for BC2 network. Our protocol significantly reduces XONN's GC-based linear layer cost, with a slight increase in nonlinear layer cost 185

Figure 7.14. Accuracy and runtime of our oblivious BNN inference, compared with contemporary research with same server-client scenario (two-party HbC). XONN [3] evaluates BNNs, whereas Cryptflow2 [5], Delphi [1], SafeNet [2], and AutoPrivacy [4] evaluate non-binary models. 186

Figure 7.15. Runtime in WAN setting with ~ 20 MBps bandwidth and ~ 50 ms network delay 187

Figure B.1. Enlarged Architecture of FASE 196

LIST OF TABLES

Table 3.1. Comparison of the No. of non-XORs of TinyGarble with Frigate 38

Table 3.2. Run-time (ms) for matrix multiplication through GC 38

Table 3.3. Inference on one image with LeNet through GC 41

Table 3.4. Run-time (ms) for the operations in TinyGarble2 41

Table 3.5. Evaluation on privacy-preserving auction. 44

Table 3.6. Evaluation on privacy-preserving voting. 44

Table 4.1. HSCD format to store the netlist 53

Table 4.2. Benchmark Functions 65

Table 4.3. Resource Utilization of FASE 66

Table 4.4. Evaluation of the Effect of the Memory Optimization 67

Table 4.5. Comparison of FASE with previous GC accelerators 68

Table 4.6. Comparison of FASE on FPGA with TinyGarble [6] on CPU 68

Table 5.1. Resource usage of one MAC unit 80

Table 5.2. Throughput Comparison of MAXelerator with state-of-the-art GC frameworks. Throughput is computed in number of MACs per sec 81

Table 5.3. Number of XOR and non-XOR gates, amount of communication and computation time for each benchmark. 82

Table 5.4. Ridge Regression Runtime Improvement 84

Table 6.1. Privacy-preserving applications presented in this chapter. 86

Table 6.2. Number of XOR and non-XOR gates in the netlists 109

Table 6.3. Timing results 110

Table 6.4. The numbers of non-XOR gates in the generated netlist for different values of threshold fraction t 131

Table 6.5. Computational time and memory utilization complexity for two different implementations of LSH. 133

Table 6.6. Timing evaluation of the authentication protocol in the two settings for different values of the threshold fraction t. 134

Table 6.7. Private set intersection (Bitwise-AND variant). 149

Table 6.8. Private set intersection (SMCS variant). 150

Table 7.1. TytaNN secure execution cost for core operations in a DNN. Here, κ is the security parameter that is set to 128. 163

Table 7.2. Cost of different phases of linear layers in COINN and previous works. N_{slot} is number of slots in vectorized HE operations. $CostMult(q)$ is cost of one scalar multiplication in $\mathcal{Z}_q$ in HE. q is cipher-text modulus which is $\sim 3\times$ larger than plain-text modulus $p \approx 2^{b_{acc}}$. 170

Table 7.3. COINN benchmarks. 174

Table 7.4. Evaluation of COINN in LAN and WAN settings. Q and C denote quantization and clustering, respectively. 179

Table 7.5. Performance comparison of COINN with best prior work. "Improv." shows the improvement in total runtime. CTF2 refers to CrypTFlow2 [5]. 180

Table 7.6. Runtime of COINN model customization and fine-tuning, normalized by the target DNN's training time on one NVIDIA Titan XP GPU. Here, Q and C denote the quantization and clustering stages, respectively. 182

Table 7.7. Evaluation on convolution layers of TytaNN with regular matrix multiplication 182

Table 7.8. Evaluation on convolution layers of TytaNN with factored matrix multiplication, $b = 16$ 183

Table 7.9. Evaluation on ReLU of TytaNN (including AS-GC conversions) 183

Table 7.10. Summary of the trained binary network architectures evaluated on the CIFAR-10 dataset 183

ACKNOWLEDGEMENTS

I would like to start with expressing my sincere gratitude to my Ph.D. advisor, Professor Farinaz Koushanfar for her continuous support and guidance. I would like to thank my committee members, Professor Andrew B. Kahng, Professor Ryan Kastner, Professor Bill Lin, and Professor Tajana Simunic Rosing for their valuable suggestions and advice during my doctoral study. I had the honor to work with and learn from Professor Ahmad-Reza Sadeghi, Professor Thomas Schneider, Dr. Rosario Cammarota, and Dr. Kristin Lauter. I was fortunate to work with several brilliant researchers. In particular, I would like to thank Dr. Ebrahim Songhori, Dr. Bita D Rouhani, Dr. Sadegh M Riazi, Sudha Yellapantula, Mohammad Samragh, Mohammad Ghasemzadeh, Huili Chen, Mojan Javaheripi, and Xinqiao Zhang. Finally, I would like to express my most sincere gratitude to my parents, my wife, my brother and sisters, and my friends for their unconditional support.

The work in this dissertation is, in part, based on the following papers.

Chapter 3, in part, has been published at (i) 2021 IEEE Security & Privacy (S&P) and appeared as: Siam U Hussain, Sadegh M Riazi, and Farinaz Koushanfar, "The Fusion of Secure Function Evaluation and Logic Synthesis", and (ii) 2020 ACM Workshop on Privacy-Preserving Machine Learning in Practice (PPMLP) and appeared as: Siam U Hussain, Baiyu Li, Farinaz Koushanfar, and Rosario Cammarota. "TinyGarble2: Smart, Efficient, and Scalable Yao's Garble Circuit", and (iii) 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) and appeared as: Sadegh M Riazi, Mojan Javaheripi, Siam U Hussain, and Farinaz Koushanfar, "MPCircuits: Optimized Circuit Generation for Secure Multi-Party Computation" (iv) 2015 IEEE Symposium on Security & Privacy (S&P) and appeared as: Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar, "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits". The dissertation author was the primary investigator of the first two papers.

Chapter 4, in full, is a reprint of the material as it appeared at 2019 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) and appeared as: Siam U Hussain

and Farinaz Koushanfar, "FASE: FPGA Acceleration of Secure Function Evaluation". The dissertation author was the primary investigator of the paper.

Chapter 5, in part, has been published at (i) 2018 ACM/IEEE Design Automation Conference (DAC) and appeared as: Siam U Hussain, Bita D Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar, "MAXelerator: FPGA Accelerator for Privacy Preserving Multiply-Accumulate (MAC) on Cloud Servers", and (ii) 2018 ACM Transactions on Reconfigurable Technology and Systems (TRETS) and appeared as: Bita D Rouhani, Siam U Hussain, Kristin Lauter, and Farinaz Koushanfar, "ReDCrypt: Real-Time Privacy-Preserving Deep Learning Inference in Clouds Using FPGAs". The dissertation author was the primary investigator of the first paper.

Chapter 6, in part, has been published at (i) 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) and appeared as: Sadegh M Riazi, Mojan Javaheripi, Siam U Hussain, and Farinaz Koushanfar, "MPCircuits: Optimized Circuit Generation for Secure Multi-Party Computation", and (ii) 2018 ACM Transactions on Design Automation of Electronic Systems (TODAES) and appeared as: Siam U Hussain, Sadegh M Riazi, and Farinaz Koushanfar, "SHAIP: Secure Hamming Distance for Authentication of Intrinsic PUFs", and (iii) 2018 ACM Transactions on Design Automation of Electronic Systems (TODAES) and appeared as: Siam U Hussain, and Farinaz Koushanfar, "P3: Privacy Preserving Positioning for Smart Automotive Systems", and (iv) 2016 ACM/IEEE Design Automation Conference (DAC) and appeared as: Siam U Hussain, and Farinaz Koushanfar, "Privacy Preserving Localization for Smart Automotive Systems", and (v) 2015 ACM/IEEE Design Automation Conference (DAC) and appeared as: Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, and Farinaz Koushanfar, "Compacting Privacy-Preserving k-Nearest Neighbor Search Using Logic Synthesis". The dissertation author was the primary investigator of (ii), (iii), and (iv).

Chapter 7, in part, has been accepted to (i) ACM Conference on Computer and Communications Security as: Siam U Hussain, Mojan Javaheripi, Mohammad Samragh, and Farinaz Koushanfar. "COINN: Crypto/ML Codesign for Oblivious Inference via Neural Networks", and

(ii) 2021 Conference on Computer Vision and Pattern Recognition (CVPR) as: Mohammad Samragh, Siam U Hussain, Xinqiao Zhang, and Farinaz Koushanfar, “On the Application of Binary Neural Networks in Oblivious Inference”. The dissertation author was the primary investigator of both papers.

This dissertation was supported, in parts, by the Office of Naval Research (ONR) (N00014-17-1-2500), National Science Foundation (NSF)/Semiconductor Research Corporation (SRC) (1619261 / 2016-TS-2690), Multidisciplinary University Research Initiative (MURI) (FA9550-14-1-0351), NSF GC@Scale (CNS-1619261), and NSF Trust-Hub (CNS-1649423) grants.

VITA

2011 Bachelor of Science in Electrical & Electronic Engineering, Bangladesh University of Engineering & Technology

2015 Master of Science in Electrical & Computer Engineering, Rice University, Houston, TX

2021 Doctor of Philosophy in Electrical Engineering (Computer Engineering), University of California San Diego

PUBLICATIONS

[1] Siam U Hussain, Mojan Javaheripi, Mohammad Samragh, and Farinaz Koushanfar. "COINN: Crypto/ML Codesign for Oblivious Inference via Neural Networks". To appear in ACM Conference on Computer and Communications Security, 2021.

[2] Siam U Hussain, Mohammad Samragh, Xinqiao Zhang, K. Huang and F. Koushanfar, On the Application of Binary Neural Networks in Oblivious Inference, IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops, May, 2021.

[3] Siam U Hussain, Sadegh M Riazi, and Farinaz Koushanfar. "The Fusion of Secure Function Evaluation and Logic Synthesis". IEEE Security & Privacy (S & P), 19(2), 2021.

[4] Siam U Hussain, Baiyu Li, Farinaz Koushanfar, and Rosario Cammarota. "TinyGarble2: Smart, Efficient, and Scalable Yao's Garble Circuit". In ACM Workshop on Privacy-Preserving Machine Learning in Practice (PPMLP), 2020.

[5] Huili Chen, Siam U Hussain, Fabian Boemer, Emmanuel Stapf, Ahmad-Reza Sadeghi, Farinaz Koushanfar, and Rosario Cammarota. "Developing Privacy-Preserving AI Systems: the Lessons Learned". In ACM/IEEE Design Automation Conference (DAC), 2020.

[6] Siam U Hussain and Farinaz Koushanfar. "FASE: FPGA Acceleration of Secure Function Evaluation". In IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2019.

[7] Sadegh M Riazi, Mojan Javaheripi, Siam U Hussain, and Farinaz Koushanfar. "MPCircuits: Optimized Circuit Generation for Secure Multi-Party Computation". In IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2019.

[8] Ebrahim M Songhori, Sadegh M Riazi, Siam U Hussain, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. "ARM2GC: Simple and Efficient Garbled Circuit Framework by Skipping". In ACM/IEEE Design Automation Conference (DAC), 2019.

[9] Siam U Hussain, Sadegh M Riazi, and Farinaz Koushanfar. "SHAIP: Secure Hamming Distance for Authentication of Intrinsic PUFs". ACM Transactions on Design Automation of Electronic Systems (TODAES), 23(6), 2018.

[10] Siam U Hussain and Farinaz Koushanfar. "P3: Privacy Preserving Positioning for Smart Automotive Systems". ACM Transactions on Design Automation of Electronic Systems (TODAES), 23(6), 2018. xvi

[11] Bita D Rouhani, Siam U Hussain, Kristin Lauter, and Farinaz Koushanfar. "ReDCrypt: Real-Time Privacy-Preserving Deep Learning Inference in Clouds Using FPGAs". ACM Transactions on Reconfigurable Technology and Systems (TRETS), 11(3), 2018.

[12] Siam U Hussain, Bita D Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar. "MAXelerator: FPGA Accelerator for Privacy Preserving Multiply-Accumulate (MAC) on Cloud Servers". In ACM/IEEE Design Automation Conference (DAC), 2018.

[13] Siam U Hussain, Mehrdad Majzoobi, and Farinaz Koushanfar. "BIST for Online Evaluation of PUFs and TRNGs". In Mark Tehranipoor, Domenic Forte, Garrett S. Rose, and Swarup Bhunia, editors, Security Opportunities in Nano Devices and Emerging Technologies, chapter 14, page 257. CRC Press, 2017.

[14] Siam U Hussain, Mehrdad Majzoobi, and Farinaz Koushanfar. "A Built-In-Self-Test Scheme for Online Evaluation of Physical Unclonable Functions and True Random Number Generators". IEEE Transactions on Multi-Scale Computing Systems(TMSCS), 2(1), 2016.

[15] Siam U Hussain and Farinaz Koushanfar. "Privacy Preserving Localization for Smart Automotive Systems". In ACM/IEEE Design Automation Conference (DAC), 2016.

[16] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits". In IEEE Symposium on Security & Privacy (S&P), 2015.

[17] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. "Compacting Privacy-Preserving k-Nearest Neighbor Search Using Logic Synthesis". In ACM/IEEE Design Automation Conference (DAC), 2015.

[18] Siam U Hussain, Sudha Yellapantula, Mehrdad Majzoobi, and Farinaz Koushanfar. "BIST-PUF: Online, Hardware-Based Evaluation of Physically Unclonable Circuit Identifiers". In IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2014.

ABSTRACT OF THE DISSERTATION

Practical, Scalable, and Efficient Privacy-Preserving Computation

by

Siam Umar Hussain

Doctor of Philosophy in Electrical Engineering (Computer Engineering)

University of California San Diego, 2021

Professor Farinaz Koushanfar, Chair

In today's data-driven world, we are conflicted with two opposing phenomena. On the one hand, collection and analysis of an enormous amount of data have resulted in rapid advances in technologies and services, especially the ones based on Artificial Intelligence (AI). On the other hand, existing and potential dangers of data misuse have created serious concern about data privacy. Privacy-preserving computation presents powerful cryptographic tools to tackle this conflict by enabling analysis on data with assurance of provable privacy guarantee. However, this capability comes with significant computation and communication overhead deterring its adoption in practical data-intensive applications. Moreover, understanding the details of the cryptographic methods often appears to be a daunting task for application developers. This

dissertation contributes towards enabling data-intensive systems with provable privacy guarantee in realistic settings. Our work addresses the challenge of practical privacy-preserving computation from three directions. First, we develop open-source frameworks with efficient and scalable execution of privacy-preserving protocols as well as a rich programming interface to abstract the details of protocol execution from the users. Second, we speed up the computations required for the protocols through custom-designed hardware platforms. Our designs include both generic and application-specific accelerators achieving a minimum of 110× improvement in throughput-per-core over the best prior art. Third, we devise several practical privacy-preserving applications including secure localization, authentication with noisy keys, and k-nearest neighbor search on private data. Our most exciting application is a mixed protocol system for privacy-preserving AI with 4.7×–14.4× speed up over state-of-the-art.

Chapter 1

Introduction

In the era of big data, ensuring privacy of sensitive user content is a standing challenge. While in many cases these data are used in scenarios that are beneficial to the data providers, misuse of the personal data may adversely affect the data owner. However, completely blocking access to the data will deprive the user of many beneficial features. The ideal solution to this conundrum is to ensure control of users over how their data is used. Especially with the recently enforced data privacy regulations worldwide, corporations have a justified interest and obligation to protect users' privacy. Even though several heuristic methodologies for privacy-preserving computing have been suggested, it is difficult to assure their resilience due to the large space of possible breaches. Solutions based on provably secure cryptographic primitives hold a promise to provide privacy guarantees within the standard security model. These primitives can be broadly categorized into two approaches differing in capabilities, computation style, and applicable scenarios: Multi-Party Computation (MPC) protocols and Homomorphic Encryption (HE). MPC allows multiple parties to jointly compute a function without revealing their respective inputs to one other. HE allows a party to compute on data encrypted by another party. Both approaches incur substantial computation and/or communication overhead compared to plaintext computation. Over the past decade, substantial research efforts, both in algorithmic optimization and efficient realization have resulted in tremendous improvement in this domain. This dissertation contributes to the effort towards making MPC-based applications practical and efficient.

The primary focus of this dissertation is enabling data-intensive systems with provable privacy guarantee in practical settings. We advance the field of privacy-preserving computation from the following three different perspectives.

- We present three open-source frameworks that help develop efficient and scalable MPC-based applications. Our first framework, named TinyGarble [6], supports developing applications based on the 2-party computation (2PC) protocol – Yao's Garbled Circuit (GC) [7]. As shown by evaluations both from us and independent researchers, TinyGarble currently provides the highest efficiency for GC-based applications. We extended a subset of the capabilities of TinyGarble to support the Beaver-Micali-Rogaway (BMR) [8] protocol – a multi-party extension of GC – in the framework named MPCircuits [9]. Recently, we open-sourced the TinyGarble2 framework [10], an upgraded version of TinyGarble, with a more user-friendly programming interface, better scalability, and enhanced security.
- We increase the execution speed of the GC protocol through hardware accelerators built on FPGA. We designed two accelerators: FASE [11], which supports privacy-preserving execution of any generic function, and MAXelerator [12], which is designed specifically for Machine Learning (ML) inference.
- We developed several real-world applications based on the frameworks. The applications include authentication with noisy keys, secure localization, k-nearest neighbor search on private data, private set intersection, and oblivious ML inference. Since in recent years most of the data-intensive operations, e.g., medical diagnosis, predictive test in phone keyboards, fraud detection, financial advice, targeted advertising, are employing ML, we devised a custom mixed protocol system, incorporating GC and protocols based on Oblivious Transfer (OT) [13] and Arithmetic Sharing (AS) [14], for oblivious inference on Deep Neural Network (DNN) and one of its special variant named the Binarized Neural Network (BNN).

In the following, we provide a brief overview of these contributions and later elaborate on them in subsequent chapters.

1.1 Efficient and Scalable MPC Frameworks

We developed three open-source MPC frameworks: TinyGarble, TinyGarble2 for the GC protocol which supports two parties, and MPCircuits for the BMR protocol which is an extension of GC supporting more than two parties. A crucial step of utilizing both of these protocols is to compile the function being computed to its Boolean logic representation. Our first framework TinyGarble presents a novel automated methodology based on powerful logic synthesis techniques for generating and optimizing compressed Boolean circuits. Moreover, TinyGarble achieves an unprecedented level of compactness and scalability by using a sequential circuit description for GC. This framework introduces new libraries and transformations, such that the sequential circuits can be optimized and securely evaluated by interfacing with available garbling frameworks. The circuit compactness makes the memory footprint of the garbling operation fit in the processor cache, resulting in fewer cache misses and thereby fewer CPU cycles. Our proof-of-concept implementation of benchmark functions demonstrates a high degree of compactness and scalability. At the time of its publication, TinyGarble improved the results of existing automated tools for GC generation by orders of magnitude; for example, TinyGarble can compress the memory footprint required for 1024-bit multiplication by a factor of 4,172, while decreasing the number of non-XOR gates by 67%. Moreover, with TinyGarble we were able to implement functions that had never been reported before, such as SHA-3. Even though several automated tools have emerged since then, TinyGarble remains the most efficient one.

Recently, we published TinyGarble2 – an upgraded version of the TinyGarble framework. TinyGarble2 provides a couple of enhancements over its predecessor. In TinyGarble, to benefit from the powerful logic synthesis techniques the user needed to describe her function in a Hardware Description Language (HDL), e.g., Verilog. TinyGarble2 provides a rich `C++` library with arithmetic and logic building blocks for developing GC-based secure applications. It thus provides the convenience of a programming language along with the efficiency of TinyGarble. The framework offers abstractions among three layers: the `C++` program, the GC back-end, and

the Boolean logic representation of the function being computed. This allows employing the most optimized versions of all pertinent components to compose arbitrary functions while providing the convenience of a programming language. In addition, TinyGarble2 provides a library with parameterized implementations of basic building blocks of Convolutional Neural Networks (CNN), which can be used to compose any privacy-preserving CNN inference. Another significant improvement is the support for the malicious security model along with the honest-but-curious model supported by TinyGarble. We evaluate TinyGarble2 on micro-benchmarks and the LeNet-5 CNN. Our evaluations show that TinyGarble2 is the only framework offering scalable execution in both security models. Moreover, TinyGarble2 performs 6× faster on LeNet-5 compared to the fastest existing scalable framework in the honest-but-curious model and is 43% more efficient in terms of memory footprint.

In the MPCircuits framework, we demonstrate that the methodology presented by TinyGarble to generate optimized Boolean logic for the two-party GC protocol is equally applicable to the multi-party BMR protocol. MPCircuits presents an end-to-end tool-chain to facilitate practical scalable MPC realization. To illustrate the practicality of MPCircuits, we design and implement a set of five functions that represent real-world MPC problems. We chose the benchmarks in a way that they inherently have different computational and communication complexities and are good candidates to evaluate MPC protocols. We also formalize the metrics by which a given protocol can be analyzed. We provide extensive experimental evaluations for these benchmarks; two of which were the *first* reported solutions in multi-party settings. Our experimental results indicate that MPCircuits reduced the computation time of MPC protocols by up to 4.2× compared to the state-of-the-art.

1.2 General Purpose Hardware Platform for Privacy-Preserving Computation

We developed two FPGA accelerators to speed up the computation required for the GC protocol: FASE and MAXelerator. They are designed to allow cloud servers to provide secure services to a large number of clients in parallel while preserving the privacy of the data from both sides. Among them, FASE is the general-purpose accelerator supporting computation of any given function through GC. General-purpose GC accelerators before FASE had low throughput due to inefficient management of resources. In FASE, we designed a pipelined architecture along with an efficient scheduling scheme to ensure optimal usage of the available resources. The scheme is built around a simulator of the hardware design that schedules the workload and assigns the most suitable task to the encryption cores at each cycle. This, coupled with optimal management of the read and write cycles of the embedded memory on FPGA, results in a minimum 2 orders of magnitude improvement in terms of throughput per core for the reported benchmarks compared to the best previous generic GC accelerator. Moreover, through application of state-of-the-art GC optimizations, we reduced the resource usage by the encryption core requires by 17%.

1.3 Co-design and Optimization of Privacy-Preserving Computation and Hardware

Our second[1] FPGA accelerator for GC, named MAXelerator, is an excellent example of the performance enhancement possible by application specific hardware platform. MAXelerator is the first hardware accelerator that is customized for privacy-preserving ML on cloud servers. Cloud-based ML is being increasingly employed in various data-sensitive scenarios. While it enhances both efficiency and quality of the service, it also raises concerns about privacy of the users' data. In this work, we show that for the majority of the ML applications, the

[1]Chronologically, MAXelerator was published before FASE. However, since it is a specialized version of FASE, we changed the order of their appearance in this manuscript.

privacy-sensitive computation boils down to either matrix multiplication, which is a repetition of Multiply-Accumulate (MAC), or the MAC itself. We design an FPGA architecture for privacy-preserving MAC to accelerate the ML computation based on the GC protocol. At the time of its publication, MAXelerator was 985× faster than the then fastest generic GC accelerator on FPGA [15]. It is also 4× faster compared to FASE for matrix multiplication. We corroborate the effectiveness of this accelerator with real-world case studies in privacy-sensitive scenarios.

1.4 Real-World Applications

We developed several real-world privacy-preserving applications based on our GC and BMR frameworks. On one hand, support for such practical applications is a testament to the powerful capabilities of these frameworks. On the other hand, development of these applications pointed out scopes of improvements in the design of the frameworks which helped us enhance them. The applications we developed include secure localization, authentication with noisy keys, and k-Nearest Neighbors Search (k-NNS) on private data, which we describe next.

Secure Localization. We designed and implemented the first privacy-preserving localization method based on provably secure primitives for smart automotive systems [16, 17]. Using this method, a car, lost due to unavailability of GPS, can compute its location with assistance from three nearby cars while the locations of all the participating cars including the lost car remain private. Technological enhancement of modern vehicles, especially in navigation and communication, necessitates parallel enhancement in security and privacy. Previous approaches to maintaining user location privacy suffered from one or more of the following drawbacks: trade-off between accuracy and privacy, one-sided privacy, and the need for a trusted third party that presents a single point to attack. The localization method presented here is one of the very first location-based services that eliminates all these drawbacks. Two protocols for computing the location are presented – one based on the two-party GC protocol and one based on the multi-party BMR protocol. The protocols exhibit trade-offs between performance and resilience against

collusion. Proof-of-concept implementation of the protocol shows that the operation can be completed within only 355 ms enabling localization of even moving cars.

Authentication with Noisy Keys. Our authentication system is named SHAIP [18] – a secure Hamming distance-based mutual authentication protocol that allows an unlimited number of authentications by employing an intrinsic Physical Unclonable Function (PUF) [19]. PUFs generate keys based on the inherent manufacturing variations of devices. These keys generally show some variations (noise) every time they are reproduced. Even though our system is designed for PUF it is also applicable to biometric authentication, which also deals with noisy keys, with little or no modification. These authentication schemes rely on secure computation of certain distance functions between the submitted and stored keys. In this work, we expose vulnerabilities of previous Hamming distance-based authentication schemes. Specifically, we show that an adversary can recover the stored key in linear (in terms of key length) number of attempts. We then present a secure authentication protocol based on the GC protocol. We show that our scheme is effective with all state-of-the-art intrinsic PUFs. The proposed scheme is lightweight and does not require any modification to the underlying hardware.

k-Nearest Neighbors Search (k-NNS) on Private Data. We introduced the first efficient, scalable, and practical method for privacy-preserving k-Nearest Neighbors Search (k-NNS) in the two-party setting based on the GC protocol [20]. The approach enables performing the widely used k-NNS in sensitive scenarios where none of the parties reveal their information while they can still cooperatively find the nearest matches. In contrast with the existing GC-based approaches that only accept function descriptions as combinational circuits, we employed sequential circuit approach presented first by the TinyGarble framework. Our proof-of-concept implementation of the k-NNS demonstrates the applicability, efficiency, and scalability of the suggested methods. Later in MPCircuits [9], we extended the design to multi-party settings executed through the BMR protocol. This implementation inherits the efficiency in run-time, though scalability in terms of memory footprint in the multi-party settings is still an open challenge.

1.5 Co-optimization of Crypto Primitives and ML Inference

The applications described above are based on either GC or its multiparty extension BMR. A recent trend in the domain of privacy-preserving computation is to adopt a mixed protocol model where the most efficient protocol for a particular operation is chosen, securely switching between different protocols when necessary. We adopt this approach in developing privacy-preserving ML inference system where the inference is performed without revealing the client's private inputs to the server or revealing server's proprietary ML weights to the client. Furthermore, we adopt a co-optimization approach where we not only design efficient cryptographic protocols for ML inference but also customize the ML model to be more amenable to the privacy-preserving computation. Our research in this field resulted in two oblivious inference systems – one for generic Deep Neural Network (DNN) and one for the Binarized Neural Network (BNN).

Our framework for oblivious DNN inference is named COINN – an efficient, accurate, and scalable framework designed for the two-party setting. In our system, to speed up the oblivious inference while maintaining high accuracy, we make three interlinked innovations in the plaintext and ciphertext domains: (i) we develop a new domain-specific low-bit quantization scheme tailored for high-efficiency ciphertext computation, (ii) we construct novel techniques for increasing data re-use in secure matrix-multiplication allowing us to gain significant performance boosts through factored operations, and (iii) we propose customized cryptographic protocols that complement our optimized DNNs in the ciphertext domain. By co-optimization of the aforesaid components, COINN brings an unprecedented level of efficiency to the setting of oblivious DNN inference, achieving an end-to-end runtime speedup of 4.7×–14.4× over the state-of-the-art. We demonstrate the scalability of our proposed methods by optimizing complex DNNs with over 100 layers and performing oblivious inference in the Billion-operation regime for the challenging ImageNet dataset.

While COINN is for generic DNN, in SlimBin, we explore the application of BNN in

oblivious inference. We make two contributions in this work. First, we devise lightweight cryptographic protocols designed specifically to exploit the unique characteristics of BNNs. Second, we present dynamic exploration of the runtime-accuracy tradeoff of BNNs in a single-shot training process. While previous works trained multiple BNNs with different computational complexities (which is cumbersome due to the slow convergence of BNNs), we train a single BNN that can perform inference under different computational budgets. Compared to the state-of-the-art in oblivious inference of non-binary DNNs, our approach reaches 3× faster inference at the same accuracy. Compared to XONN [3], the state-of-the-art in oblivious inference of binary networks, we achieve 2×-12× faster inference while obtaining higher accuracy.

Chapter 2

Background

In this chapter, we provide a brief overview of the privacy-preserving primitives employed in this work. We also review the basic building blocks of Deep Neural Networks (DNN).

2.1 Notations

Throughout this manuscript, we represent scalars with lowercase x, vectors with bold lowercase $\mathbf{x}$, 2-dimensional matrices with uppercase X, and higher-order tensors with bold uppercase letters $\mathbf{X}$. Element selection is denoted by square brackets $\mathbf{x}[i]$ and $x\langle i\rangle$ denotes the i-th bit of scalar x. $\mathbf{0}$ denotes a vector/matrix/tensor with all the entries set to 0. We denote the computational security parameter with κ and set it to 128 following recent works [21, 6, 22].

2.2 Secure Multi-Party Computation (MPC)

Secure Multi-Party Computation (MPC) is a set of cryptographic protocols that allow two or more parties to jointly compute a function on their private inputs without revealing the inputs to each other. In MPC, the intermediate results of computation are shared between the computing parties such that no single party can learn the actual value. The efficiency of performing various arithmetic and logical operations through an MPC protocol depends on the employed sharing schemes. In the end, one or more parties learn the output. The security of MPC is analyzed in the following security models.

- **Honest but Curious:** In this model, all parties follow the protocol honestly yet may try to learn additional information about the other partys' data from the information at hand.
- **Malicious**: In this model, any party can deviate from the protocol to learn more information about the other party's data or to produce incorrect results.

2.3 Oblivious Transfer

Oblivious Transfer (OT) [13] is a cryptographic protocol between a receiver Alice and a sender Bob. OT allows Alice to choose and receive one from a set of messages provided by Bob without revealing her choice. In a 1-out-of-2 OT protocol (OT_λ^2), Bob holds a pair of λ-bit messages $\{\mu_0, \mu_1\} \in \{0,1\}^\lambda$; Alice holds a choice bit $\sigma \in \{0,1\}$ and obtains μ_σ without revealing σ. Alice learns nothing about the other message $\mu_{1-\sigma}$.

OT employs public-key cryptography which is costly. An extension of this protocol, called OT extension [23], allows performing a large number of OT_λ^2 with a fixed number of base OTs and linear (in terms of the number of OT_λ^2) number of less expensive private-key operations. Moreover, by following [24], the majority of the computation of OT can be performed in the offline phase, enabling a very fast online phase.

In this work, we employ two versions of OT extension: random OT (ROT_λ^2) and correlated OT (COT_λ^2) [25]. In ROT_λ^2, instead of choosing his messages, Bob receives random messages $\{\mu_0, \mu_1\}$ and Alice receives μ_σ. The communication cost of one ROT_λ^2 is a κ-bit message embedding the selection bit σ from Alice to Bob. Note that the cost of ROT_λ^2 is independent of the message length λ. In COT_λ^2, Bob chooses a correlation function $\varphi(\mu)$ and receives a random message μ. Alice receives μ if $\sigma = 0$ and $\varphi(\mu)$ if $\sigma = 1$. The communication cost of one COT_λ^2 is the cost of one ROT_λ^2 plus a λ-bit message from Bob to Alice, i.e., $\kappa + \lambda$.

2.4 Yao's Garbled Circuit

Yao's Garbled Circuit (GC) [7] is currently the most efficient 2PC protocol. It allows two parties Alice and Bob to jointly compute a function $c = F(a, b)$ on their private inputs a from Alice and b from Bob. At the end of the protocol, one or both of them learn the output c. In GC, a function F is represented as a Boolean logic circuit, called *netlist*, consisting the logic gates AND: $(\alpha, \beta, \gamma, \wedge)$ and XOR: $(\alpha, \beta, \gamma, \oplus)$, where α and β are the two input wires and γ is the output wire of a gate. The values associated with the wires α, β, and γ are x, y, and z respectively. Note that GC supports any 2-input 1-output logic gate, along with the NOT gate. For simplicity, we focus on these two gates here. The sets of wires associated with Alice's input, Bob's input, and the output are called A, B, and C, respectively.

Alice, the *garbler*, garbles the circuit as follows. For each wire, w in the netlist, she assigns two κ-bit random labels L_w^0 and L_w^1, corresponding to the values 0 and 1, respectively, and a 1-bit random mask λ_w. If the value of a wire is v, then the *masked value*, observed by Bob, the *evaluator*, is $\hat{v} = v \oplus \lambda_w$. For each gate in the netlist, Alice generates a garbled truth table by encrypting the output labels with the corresponding input labels. Let H be a hash function modeled as a random oracle. Each row of the garbled truth table GT of a gate (α, β, γ, $t \in \{\wedge, \oplus\}$), is computed as

$$GT(2\hat{x} + \hat{y}) = H(L_\alpha^x \parallel \hat{x}, L_\beta^y \parallel \hat{y}) \oplus L_\gamma^z \parallel \hat{z} \tag{2.1}$$

Alice sends the garbled tables, the labels corresponding to her input values, and the masked values of each input wire to Bob. Bob obtains the labels corresponding to his input values obliviously through OT_1^2. At this point, for each input wire, $w \in A \cup B$ with value v, Bob holds the label L_w^v. For each gate (α, β, γ, $t \in \{\wedge, \oplus\}$) connected to the input wires, he holds $L_\alpha^x \parallel \hat{x}$ and $L_\beta^y \parallel \hat{y}$ and can decrypt only one row of the gabled table to obtain $L_\gamma^z \parallel \hat{z}$. He then uses these labels to compute the labels and masked values of the outputs of the subsequent gates. This way,

the final output that Bob can learn is the masked values $\hat{v}$ of the output wires $w \in C$. In the last step of the GC protocol, Alice sends the masks λ_w of the output wires $w \in C$ and Bob computes the actual value v of each output wire as $v = \hat{v} \oplus \lambda_w$. This step can be reversed to let Alice learn the final output.

2.4.1 Garbled Circuit Optimizations

The GC protocol has gone through several optimizations. We briefly discuss the most important ones here.

(i) Point and Permute [8]. According to this optimization, the label of each wire is appended by a select bit, such that the select bits for the two labels of the same wire are inverse of each other. Even though the select bits are public, the association between select bits and semantic value of the wire is random and private to the garbler. Besides allowing the use of more efficient encryption, it also makes the evaluation simpler since the evaluator can simply decrypt the appropriate row based on the public select bits of the wire labels.

(ii) Free-XOR [26]. In this optimization, the XOR gates do not require garbling, i.e., computation of the hash function or communication of the garbled tables. Alice generates a random κ-bit key Δ which is known only to her. For each wire w, she generates the label L_w^0 and sets $L_w^1 = L_w^0 \oplus \Delta$. Moreover, for each XOR gate $(\alpha, \beta, \gamma, \oplus)$, the masks are computed as $\lambda_\gamma = \lambda_\alpha \oplus \lambda_\beta$. With this convention, during garbling, Alice computes the 0-label for the output wire of an XOR gate as $L_\gamma^0 = L_\alpha^0 \oplus L_\beta^0$. During evaluation, Bob computes $L_\gamma^z \parallel \hat{z} = L_\alpha^x \parallel \hat{x} \oplus L_\beta^y \parallel \hat{y}$.

(iii) Row Reduction [27]. In this optimization, the size of the garbled tables for non-XOR is reduced by 25%. Instead of generating the label for the output wire of a gate randomly, it is computed as a function of the labels of the inputs such that the first entry of the garbled table becomes all 0s and no longer needs to be sent.

(iv) Half Gate [28]. In this optimization, each non-XOR gate is broken into two half-gates, for which one party knows one input. It employs both free-XOR and row reduction such that each

half-gate can be garbled with single encryption. As a result, the size of the non-XOR gate truth table is reduced by a further 25%.

(v) Fixed-key Block Cipher [21]. This optimization allows efficient garbling and evaluation non-XOR gates using fixed-key AES with a unique identifier for each gate. The output label L^z_γ is encrypted with the input labels L^x_α and L^y_β using the following encryption function

$$E(L^x_\alpha, L^y_\beta, T, L^z_\gamma) = \pi(K) \oplus K \oplus L^x_\alpha, \quad (2.2)$$

where $K = 2L^x_\alpha \oplus 4L^y_\beta \oplus T$, π is a fixed-key block cipher (instantiated with AES), and T is a unique gate identifier.

Among the optimizations discussed above, only free-XOR concerns the netlist generation process. A *GC-optimized* netlist implies that it has the least number of non-XOR gates.

2.4.2 Extension of GC for the Malicious Security Model

Yao's GC is proven to be secure in the honest-but-curious security model [29, 21]. While this model is sufficient in a large number of applications, certain applications, e.g., authentication, the Function as a Service (FaaS), require security in the malicious security model. Among various protocols that are secure in the malicious security model, the most efficient realization to date is the Authenticated Garbling protocol presented in [22]. The critical enhancements in this protocol over the semi-honest version are the following: (*i*) to XOR-share the mask bit itself for each wire between Alice and Bob, (*ii*) to authenticate their shares using Message Authentication Codes (MAC) - to ensure that none of them can alter their respective shares of the mask bit during the protocol execution, and (*iii*) to compute (by both parties) the garbled tables in a distributed manner, where each wire has two sets of labels - one generated by each party. Since the parties compute the garbled tables together, one OT^2_1 is required per gate, as opposed to per input wire as in the honest-but-curious model. The garbled circuit is *authenticated* in the sense that neither Alice nor Bob can change the logic of the circuit and/or the protocol without being caught.

The protocol presented in [22] is further optimized in [30]. The most notable enhancements by [30] are compatibility with half gate optimization [28] and avoiding communication of the MACs for each row of the garbled tables.

2.5 Beaver-Micali-Rogaway

Beaver-Micali-Rogaway (BMR) [8] is a multi-party extension of Yao's GC, supporting more than two parties. All the parties jointly participate in the preparation of the garbled circuit, and no subset of colluding parties can learn any value internal to the netlist. The function is of the form $c = F(a_0, a_1, ..., a_{n-1})$, where there are n parties involved and a_i is the private input of the i-th party. This protocol has two main phases: garbling and evaluation. In the first phase, all parties jointly create the *garbled* version of the circuit. In the second phase, each party receives partial information from other parties and begins to evaluate the circuit locally. The garbling phase is usually the most costly stage in the protocol execution. However, since it is independent of the actual inputs from the participating parties, it can be pre-computed in advance.

Garbling. In this phase, all parties assign two random labels for every wire in the circuit, one for semantic value zero and one for semantic value one. $L^x_{w,i} \in \{0,1\}^\kappa$ denotes random label of wire w for the semantic value $x \in \{0,1\}$ held by party P_i $i = 1...n$ where n is the total number of parties. For each gate, parties encrypt output labels using H, a double-key pseudorandom function, and use two input labels as keys. Consider a gate $(\alpha, \beta, \gamma, t \in \{\wedge, \oplus\})$. The values associated with the wires α, β, and γ are x, y, and z respectively. In the case of an AND gate: $(\alpha, \beta, \gamma, \wedge)$, output label for semantic value 1 (L^1_γ) is encrypted using the two input labels of semantic value 1 ($L^1_{\alpha,i}$ and $L^1_{\beta,i}$). Since there are four possible input combinations for any two-input Boolean gate, parties create four different encryptions of the correct output label and their corresponding input keys. The collection of all four encrypted values is called a *garbled table*. More precisely, for

every $\alpha, \beta \in \{0,1\}$, the output label for $\gamma \in \{0,1\}$ is encrypted as

$$\left\{\left(\bigoplus_{i=1}^{n} H_{L^{x}_{\alpha,i},\, L^{y}_{\beta,i}}(g \circ j) \oplus L^{j}_{\gamma,z}\right)\right\}_{j=1}^{n} \tag{2.3}$$

where g is the unique ID number for a gate and $\circ$ denotes concatenation operation. In order to mask the relationship between labels and actual semantic values, each party also assigns a *permutation bit* λ_w^i and sets $\lambda_w = \oplus_{i=1}^{n} \lambda_w^i$. All four encrypted values are permuted according to permutation bits.

Evaluation. Given the collection of n keys for each input wire, all parties can decrypt one row of each garbled table (those connected to input gates) and generate the output keys of those gates. The evaluation process continues until output gates are reached. Therefore, the evaluation process can be computed locally once each party has the correct combination of all n keys for all input gates. Note that none of the intermediate values are revealed to any party. The semantic value of each wire is XOR-shared among all parties. All labels are unintelligible by themselves. At the end of the protocol, each party only sends her share of the output wires' labels such that everyone can locally compute the plaintext output result. Please see [31] for more detailed explanation.

Free-XOR Optimization. Kolesnikov et al. [26] proposed a method that eliminates the need for creating garbled tables for XOR gates, rendering them almost free of cost. To utilize this technique, each party P_i needs to create a one-time random number $\Delta_i \in \{0,1\}^{\kappa}$. Same as before, $L^0_{w,i}$ is generated randomly but $L^1_{w,i}$ is set to $\Delta_i \oplus L^0_{w,i}$ for every wire. Due to this correlation of labels, the output label of each XOR gate can be computed by XORing the two input labels without any communication between parties.

2.6 Arithmetic Sharing

We denote the arithmetic share (AS) [14] of an integer x between two parties Alice and Bob as $[\![x]\!]$. For b-bit arithmetic sharing, $[\![x]\!] = [\![x]\!]_A + [\![x]\!]_B \bmod 2^b$, where $[\![x]\!]_A$ is held by Alice and $[\![x]\!]_B$ is held by Bob with $[\![x]\!], [\![x]\!]_A, [\![x]\!]_B \in \mathbb{Z}_{2^b}$. To reveal a shared variable, Alice and Bob

send their respective shares to each other and reconstruct the actual value locally. All operations on arithmetic shared values are performed in ring $\mathbb{Z}_{2^b}$, i.e., operations are $\mathsf{mod}\ 2^b$. For simplicity we do not explicitly mention $\mathsf{mod}\ 2^b$.

2.6.1 Addition and Multiplication in AS

In AS, addition of shared variables is free since each party can locally add their shares without communication. Multiplication can be performed through COT following [32]. Let us consider the scalar product $[\![z]\!] = [\![w]\!]_A [\![x]\!]_B$. For each bit $i \in [b]$, Alice and Bob engage in one COT_b^2. Bob acts as the sender with the correlation function $\phi(\mu_i) = \mu_i + [\![x]\!]_B * 2^i$ and receives μ_i. Alice acts as the receiver with choice bit $\sigma_i = [\![w]\!]_A \langle i \rangle$ and receives $\mu_{\sigma_i} = \mu_i + \sigma_i [\![x]\!]_B * 2^i$. Alice and Bob then compute $[\![z]\!]_A = \sum_{i=0}^{b-1} \mu_{\sigma_i}$ and $[\![z]\!]_B = -\sum_{i=0}^{b-1} \mu_i$, respectively. The communication cost of computing each multiplication is $b(\kappa + b)$.

2.7 Machine Learning Layers

Contemporary DNNs comprise two classes of layers: linear (convolution, fully-connected, batch normalization, and average-pooling) and non-linear (max-pooling and ReLU). We briefly explain commonly used layers in each category.

Convolution. A convolution layer (CONV) is a linear operation $F(\mathbf{X}, \mathbf{W}, \mathbf{b}) : \mathbb{R}^{C \times D_1 \times D_1} \to \mathbb{R}^{M \times D_2 \times D_2}$, where $\mathbf{X} \in \mathbb{R}^{C \times D_1 \times D_1}$ is the 3-way input tensor, $\mathbf{W} \in \mathbb{R}^{M \times C \times k \times k}$ is the 4-way weight tensor, $\mathbf{b} \in \mathbb{R}^M$ is the bias vector, and $\mathbf{Y} \in \mathbb{R}^{M \times D_2 \times D_2}$ is the 3-way output tensor. The plaintext operation of CONV can be represented as a matrix-multiplication followed by bias addition $Y = W \cdot X + \mathbf{b}$ where $W \in \mathbb{R}^{M \times N}$ is achieved by reshaping the original 4-way tensor into a $2D$ matrix and $X \in \mathbb{R}^{N \times L}$ is formed by sliding through the original 3-way tensor and vectorizing the corresponding windows into matrix rows. Each element of the output is computed via a vector dot product (VDP) and the total number of VDPs required for the matrix-multiplication is $M \times L$.

Fully-Connected. The fully-connected (FC) layer takes a vector $\mathbf{x} \in \mathbb{R}^N$ and generates

the output vector $\mathbf{y} = W \times \mathbf{x} + \mathbf{b}$ where $W \in \mathbb{R}^{M \times N}$ and $b \in \mathbb{R}^M$ are the weight and bias, respectively. Similar to CONV, the matrix-vector multiplication consists of M VDPs between rows of W and $\mathbf{x}$.

Batch Normalization. Batch normalization (BN) is a common linear operation applied on the output of CONV layers to adjust the range of numerical values. At test time, BN computes $\mathbf{y}_i^{(BN)} = \alpha_i \mathbf{y}_i + \beta_i$, where α_i and β_i are constant scalars, $\mathbf{y}_i$ is one row of the output $Y \in \mathbb{R}^{M \times L}$ from the preceding CONV, and $\mathbf{y}_i^{(BN)}$ is the corresponding row after BN.

Pooling. Contemporary DNNs include two forms of pooling layers, namely max-pooling (MP) and average-pooling (AP). These layers extract $k \times k$ windows from the input $\mathbf{X} \in \mathbb{R}^{C \times D_1 \times D_1}$ and compute the average or the maximum value in the enclosed window as the output. Assuming the $k \times k$ windows are non-overlapping, pooling layers reduce data dimensionality from $C \times D_1 \times D_1$ to $C \times \frac{D_1}{k} \times \frac{D_1}{k}$.

ReLU. This layer often follows a linear layer to introduce non-linearity in the model. A ReLU operation simply replaces negative inputs with zero and keeps positive values intact.

Chapter 3

Efficient and Scalable MPC Frameworks

3.1 Overview

Two standing challenges that users face while developing privacy-preserving systems through MPC protocols are: (i) efficient realization of the different steps of the protocol as well as its various optimizations, and (ii) understanding the cryptographic details that ensure the data privacy. The first complication often results in a high inefficiency in the protocol execution. The latter is even more critical, triggering possible security breaches if the protocol is not followed properly. In this chapter, we present three MPC frameworks providing an end-to-end solution that bridges the gap between usability and secure realization of MPC protocols. Our main focus here is on MPC protocols based on Boolean circuits, i.e., GC and BMR protocols. Later in Chapter 7, we present efficient realization of MPC protocols based on arithmetic circuits.

The practicality of MPC frameworks primarily depends on the following properties: (i) fast protocol execution, (ii) automatic generation of Boolean logic optimized for MPC, (iii) scalability in terms of memory footprint, (iv) reliability, and (v) a rich programming paradigm. The first two properties received a lot of attention from the researchers over the past years. Various optimizations, to both the protocol [26, 27, 28, 21] and the automatic generation of netlists [6, 33, 34, 29] have resulted in orders of magnitude reduction in the run-time. Recently, with the surge in the development of practical privacy-preserving systems, scalability, reliability, and a rich programming paradigm are becoming increasingly important. However, while

existing frameworks focus on a subset of these properties, none of them demonstrate the best possible performance on all the properties. Our first framework TinyGarble [6] provides the best performance in **Boolean logic generation** and **reliability** for the 2PC GC protocol. We extend these contributions to the BMR protocol for MPC in our subsequent work named MPCircuits [9]. Our most recent framework TinyGarble2 [10] provides the best performance in **scalability** in addition to a **rich programming paradigm** for GC, while also inheriting the contributions of its predecessor TinyGarble. In terms of **fast execution**, both TinyGarble and TinyGarble2 implement the most recent optimizations to the GC protocol. MPCircuits is interfaced to the BMR framework presented in [31] which implements the most recent optimizations to the BMR protocol. In the following, we elaborate on these properties.

3.1.1 Automatic Generation of Optimized Boolean Logic

As explained in Section 2.4.1, an MPC-optimized netlist implies that it has the least number of non-XOR gates. The research on optimizing Boolean logic has followed two parallel paths. On the one hand, several custom compilers supporting (or designing) various programming languages have emerged for addressing this issue. However, such custom compilers have been shown to have reliability issues and limitations in global optimization [33]. On the other hand, techniques for interpreting a behavioral description in a Boolean format are widely researched for designing digital integrated circuits (IC). Design automation for the purpose of IC design is a true engineering success story; the tools have enabled us to scale our chips to billions of gates to support complicated tasks. There was a wide gap between the capabilities of conventional IC design automation tools to compile sophisticated functions and what could be achieved by the custom MPC compilers.

Our first GC framework, named TinyGarble, bridges this gap by formulating GC netlist generation as an atypical circuit synthesis task that can be addressed and scaled with standard IC logic synthesis tools. TinyGarble presents a synthesis library and a set of optimization goals to generate the optimized Boolean logic netlists for GC by using logic synthesis tools.

To date, TinyGarble remains the most efficient netlist generation tool as verified by the study of Frigate [33]. In our subsequent work, named MPCircuits [9], we extend the methodology presented in TinyGarble to the BMR protocol, which supports more than two parties.

3.1.2 Rich Programming Paradigm

A limitation of TinyGarble is the lack of a rich programming paradigm; the users have to describe the function in Verilog HDL. Our most recent GC framework, named TinyGarble2, combines efficient circuit generation with a rich programming language along with a more flexible protocol execution flow. TinyGarble2 provides a rich C++ library with common arithmetic and logical building blocks, the netlists of which are generated by logic synthesis tools. Moreover, it provides an automated toolchain that allows users to generate and incorporate any custom circuit into the program interface of the framework.

The TinyGarble2 framework is developed in three layers. The first layer includes the pre-compiled (through TinyGarble) GC-optimized netlist. The second layer is the protocol execution back-end that takes a netlist as input and executes the protocol. The abstraction of netlist generation and protocol execution allows TinyGarble2 to benefit from all the existing netlist optimization techniques as well as support any user-defined bit-width for the variables. Users can access this layer directly and run any combination of netlists. The third layer, which is a program interface to the GC back-end, allows more convenience as it supports using common arithmetic and logical building blocks (e.g., $=$, $+$, $-$, $\times$, $\div$, %, $\sqrt{}$, if-else, $<$, $>$, &, |, $\wedge$ etc.) to develop applications. The GC back-end internally manages the secure transfer of shares among consecutive operations (netlists) according to user-defined flags. The software distribution of TinyGarble2 includes the most optimized netlists to date for the building blocks generated by TinyGarble.

One exciting feature of TinyGarble2 is that its GC back-end has two versions for two different security models: honest-but-curious and malicious. These two versions are transparent to the program interface. This feature allows the user to execute the same netlist/program in the

security model of their choice without re-writing or recompiling their code. To the best of our knowledge, only one of the existing frameworks- EMP-Toolkit [35] supports GC execution in the malicious model. However, it does not provide a program interface and lacks scalability in terms of memory footprint. As demonstrated by our experimental results, TinyGarble2 supports scalable execution in both security models.

3.1.3 Scalability in Terms of Memory Footprint

The allocation of memory is one of the primary limiting factors in the application of GC to practical size problems such as Convolutional Neural Network (CNN) inference, which has been possible only by heavily specializing the native input data types, e.g., by binarization of either activations or weights. For instance, LeNet-5 [36] – a small Convolutional Neural Networks (CNN) – requires $341k$ Multiply-Accumulate (MAC) operations per inference [37]. Even for moderately large CNNs such as VGG-16 [38], the number of MACs reaches billions. Each MAC, in turn, requires $O(b^2)$ gates for a b-bit fixed-point representation. As a result, the secure execution of an entire CNN as a single netlist without compromising privacy (e.g., without revealing any intermediate results to any party) results in an unmanageably large memory footprint. Scalability is the primary limiting factor in the existing GC frameworks as explained in the following.

A number of the existing GC frameworks [33, 34, 32, 39, 29] support `C` or `C++` or subsets of them. Among them, Frigate [33] and CBMC-GC [34] focus on generating the optimized netlist which can later be executed through any GC back-end. This approach requires that the entire netlist is generated before the protocol execution, thus increases the peak memory and hurts the scalability. Even though ABY [32] provides the execution back-end, it also generates the entire netlist ahead of the execution resulting in scalability issues.

The PCF [29] and TinyGarble [6] frameworks partially solve the scalability issue. Through run-time loop unrolling and *sequential* GC, respectively, they ensure that not all the garbled gates reside in the memory at the same instance. However, this unrolling process slows down

the protocol execution by PCF. Moreover, its netlists are 50-80% less optimized compared to the recent frameworks. In the sequential GC by TinyGarble, the same netlist is executed through the protocol for a pre-specified number of cycles. However, TinyGarble only supports homogeneous loops, while most of the practical problems require heterogeneous loops, i.e., loops where possibly different netlists are executed at every cycle.

To the best of our knowledge, two existing frameworks provide scalability in terms of memory footprint while also allowing the comfort of a programming language – Obliv-C [39] and EMP-Toolkit [35]. Obliv-C is a custom compiler that supports GC-based privacy-preserving computation.Being an extension of the `gcc` compiler, Obliv-C inherits its memory management procedures. However, this inheritance comes with limitations. First, Obliv-C does not support abstraction between netlist generation and protocol execution, therefore cannot use the best netlist generation tools. Run-time netlist generation results in an additional slowdown in execution. Moreover, Obliv-C does not allow logic level optimizations, hence it misses noteworthy optimization opportunities for certain functions (e.g., Hamming Distance which is widely used in secure authentication [18, 40, 41]). Second, Obliv-C only supports a subset of native data types in `C`, i.e., integers with bit-widths of 16, 32, and 64 bits, incurring additional overhead for applications that require more flexible data representation. Third, a custom compiler may result in unreliable binaries.

The EMP-toolkit [35] also presents a similar framework. Its execution engine is faster than both Obliv-C and ABY. While it supports arbitrary bit-width, it does not support operation involving variables with mismatched bit-widths that results in additional overhead in practical applications (this is also true for Obliv-C). An important feature of this framework is the support for the malicious security model while other frameworks target only the honest-but-curious model. However, their maliciously secure framework does not have a programming interface. A user can generate a netlist using the interface from the honest-but-curious framework and use it with the malicious one but will suffer from scalability issues similar to ABY.

3.1.4 Reliability

Beside efficient Boolean logic representation, one crucial advantage of industrial logic synthesis tools is benefiting from their reliability. In contrast to the custom GC synthesis tools, the industrial tools go through rigorous quality control. The study by the Frigate framework [33] in 2016, found reliability issues in all the existing GC frameworks, except TinyGarble. Since TinyGarble2 uses the netlists generated by TinyGarble, we expect it to inherit reliability features of TinyGarble.

3.1.5 Evaluation Results

To demonstrate the enhanced efficiency and scalability of TinyGarble2, we designed a `C++` library for privacy-preserving CNN inference through the GC protocol. The library includes parameterized implementations of the CNN layers (e.g., convolution layer, fully connected layer, ReLU, Maxpool, ArgMax, etc.) that can be plugged in to compose any CNN model. An exclusive feature of TinyGarble2 – computation involving variables with mismatched bit-widths, allows the most optimized implementation of the different layers of the CNN. We built the CNN model LeNet-5 [36] to run inference on the MNIST dataset [42] with the CNN library of TinyGarble2. In our implementation, privacy-preserving inference on one input image requires 58s with a peak memory usage of 46MB. This is 6×, 18×, respectively faster and 43%, 64%, respectively more memory efficient compared to EMP-toolkit and Obliv-C. Moreover, TinyGarble2 is the only framework with the scalability to run CNN inference in the malicious model. In addition to CNN, we evaluate our framework on micro-benchmarks and compare the run-time and memory usage with EMP-Toolkit, Obliv-C, and ABY – the existing end-to-end frameworks that support developing with `C/C++`. Our evaluations show that TinyGarble2 is the only framework that is scalable in terms of memory footprint both in honest-bit-curious and malicious settings.

3.1.6 Summary of Contributions

As mentioned, TinyGarble2 inherits all the contributions of TinyGarble and improves upon them. Therefore, we summarize the contributions of the TinyGarble2 framework here.

- We present an end-to-end GC framework with the following properties:
  - A GC-execution back-end with all recent optimizations.
  - A C++ library with common arithmetic and logical building blocks.
  - Scalable execution through secure transfer of shares.
  - Ease of programming with any generic C++ compiler.
- We present a C++ library for privacy-preserving CNN inference through the GC protocol.
- We support GC execution of the same program through either of honest-but-curious or malicious security models.
- We demonstrate the enhanced execution speed and scalability of TinyGarble2 over existing GC frameworks for both micro-benchmarks and practical systems.

In the following, we first describe how we adapt the logic synthesis tools to generate optimized Boolean logic for MPC. Next, we outline the execution flow of the GC back-end of TinyGarble2 followed by its program interface. Finally, we present the evaluation results demonstrating the superior performance of our MPC frameworks.

3.2 Netlist Generation through HDL Synthesis

As described in Section 2.4.1, Yao's protocol requires the function to be represented as a Boolean circuit. Previous work like FairPlay [43] and WYSTERIA [44] used custom-made languages to describe a function and generate the circuit for GC operations. In our TinyGarble framework, the user may describe a function in a standard HDL like Verilog or VHDL. She may also write the function in a high level language like C/C++ and convert it to HDL using a HLS tool. TinyGarble uses existing HDL synthesis tools to map an HDL to a list of basic binary gates. In digital circuit theory, this list is called a *netlist*. The netlist is generated based on various

constraints and objectives such that it is functionally equivalent to the HDL/HLS input function. Exploiting synthesis tools helps to reduce both number of non-XOR gates in the circuit and the garbling time while also making the framework easily accessible.

3.2.1 Synthesis Flow

In the first step, a synthesis converts functional description of a circuit into a structural representation consisting of standard logical elements. Then, it converts this structural representation into a netlist specific to the target platform. In both steps, the synthesis tool works under a set of user defined constraints/objectives like minimizing the total delay or limiting the area. In the following, we describe the details of these two steps and how we manipulate the synthesis tools in each of the steps to generate optimized netlists for GC.

Synthesis library. The first step in the synthesis flow is to convert arithmetic and conditional operations like add, multiply, and if-else to their logical representations that fits best to the user's constraints. For example, the sum of two N-bit numbers can be replaced with an N-bit ripple carry adder in case of area optimization or an N-bit carry look ahead adder in case of timing optimization. A library that consists of these various implementations is called a *synthesis library*. We develop our own synthesis library that includes implementations customized for SFE. In this library, we build the arithmetic operations based on a full adder with one non-XOR gate [45] and conditional operations based on a 2-to-1 multiplexer (MUX) with one non-XOR gate [26].

Technology library. The next step is to map the structural representation onto a *technology library* to generate the netlist. A technology library contains basic units available in the target platform. For example, tools targeting Field Programmable Gate Arrays (FPGAs) like Xilinx ISE or Quartus contain Look-Up Tables and Flip Flops in their technology libraries, which form the architecture of an FPGA. On the other hand, tools targeting Application Specific Integrated Circuits (ASICs) like Synopsys DC, Cadence, and ABC, may contain a more diverse

collection of elements starting from basic gates like AND, OR, etc., to more complex units like FFs. The technology library contains logical descriptions of these units along with performance parameters like their delay and area. The goal of the synthesis tool in this step is to generate a netlist of library components that best fit the given constraints. For HDL synthesis, we use tools targeting ASICs as they allow more flexibility in their input technology library. We design a custom technology library that contains 2-input gates as required by the front-end GC tools. We set the area of XOR gates to 0 and the area of non-XOR gates to a non-0 value. By choosing area minimization as the only optimization goal, the synthesis tool produces netlists with the minimum possible number of non-XOR gates.

An additional feature of our custom technology library is that it contains non-standard gates (other than basic gates like NOT, AND, NAND, OR, NOR, XOR, and XNOR) to increase flexibility of mapping process. For example, the logical functions $F = A \wedge B$ and $F = (\neg A) \wedge B$ requires equal effort in garbling/evaluation. However by using only standard gates, the second function will require two gates (a NOT gate and an AND gate) and store one extra token for $\neg A$ in the memory. We include four such non-standard gates with an inverted input in our custom library.

For synthesis of sequential circuits, the technology library includes memory elements. These elements can be implemented as FFs which are connected to a clock signal. Although in conventional ASIC design FFs are typically as costly as four AND gates, in our GC application, FFs do not have any impact on the garbling/evaluation process as they require no cryptographic operations. Therefore, we set the area of FFs to 0 to show its lack of impact on computation and communication time of garbling/evaluation. Moreover, we modify our FFs such that they can accept an initial value. This helps us remove extra MUXs in standard FF design for initialization.

3.2.2 Offline Circuit Synthesis

In TinyGarble, we use HDL synthesis tools in an offline manner to generate a circuit for a given functionality. This offline synthesis followed by a topological sort provides a ready-to-use

circuit description for any GC framework. This approach, unlike online circuit generation, does not require misspending time for circuit generation during garbling/evaluation. It also enables the use of beneficial synthesis optimization techniques that were previously infeasible for online generation. Moreover, the synthesis tools have a global view of the circuit, unlike previous work that manually optimized small modules of the circuit. This allows more effective optimization for any arbitrary function and set of constraints.

However, the offline approach has certain limitations when it comes to generating circuits for extremely large functions. Fortunately, the sequential description helps to overcome most limitations as it generates more compact circuits. Sequential circuits are radically smaller than combinational ones with the same functionality. This property allows synthesis tools to perform more effective circuit optimization. Moreover, the compatibility of our sequential descriptions with standard synthesis tools simplifies the workflow of circuit generation for SFE applications.

3.2.3 Adaptation to BMR and GMW protocols

The GC execution of TinyGarble supports secure two-party computation in the honest-but-curious model. However, the capability of its netlist generation tool-chain goes well beyond that. The primary target of the netlist generation methodology presented in the previous section is to take advantage of the free-XOR optimization. Fortunately, equivalent optimizations are available in a number of related MPC protocol based on the Boolean logic representation of a given function.

In our work titled MPCircuits [9], we present the first automated methodology to generate Boolean circuits, customized for the BMR protocol, which is an extension of GC with support for more than two parties. This work adapts the interfaces to the synthesis and technology libraries of TinyGarble to accept inputs from multiple parties. We also develop five practical privacy-preserving applications, namely, stable matching, voting, auction, set intersection and k-nearest neighbor search on this framework and report their performance on the implementation of the BMR protocol presented in [46]. Each benchmark captures a different set of requirements

and domains, which ensures the applicability of MPCircuits to diverse scenarios. Out of the five benchmarks, we elaborate on the private set intersection in Section 6.5.

Inspired by TinyGarble, Demmler et. al. employed logic synthesis tools to generate netlists for the GMW protocol [47]. Since the round complexity of the GMW protocol depends on the depth of the netlist, they developed a tool-chain to optimize the netlist not only for size but also for depth. Their work showed a reduction of depth by up to 14% even over manually optimized netlists.

GC, BMR, and GMW protocols involve logic gates whose functionalities are fixed (e.g., `AND`, `OR`). Therefore, TinyGarble, MPCircuits and the work in [47] employ ASIC synthesis tools. Dessouky et. al. [48] introduce protocols involving lookup tables (LUTs) which can be programmed to realize arbitrary functions. To generate the Boolean circuits, this work employs multi-input LUT-based synthesis tools which form the core of synthesis for FPGAs.

3.3 Execution Flow of the GC Back-end

The TinyGarble2 framework is developed in three layers. The first layer includes the pre-compiled GC-optimized netlist. The second layer is the back-end that executes the GC protocol on any given set of netlists. The third layer provides an interface between the back-end and arithmetic/logical function building blocks. In this section, we elaborate on the execution flow of the back-end and explain how it helps the realization of scalable privacy-preserving applications. The users can directly access this level from the command line or using a configuration file (please see Appendix A for the command-line options). However, it is more convenient to use the functionalities from the second layer (described in the next section) to develop applications.

3.3.1 Function Composition Formats

The GC back-end allows compositions of the functions with the three formats in Eqs 3.1 - 3.3 as well as any hybrid combinations. The first format with homogeneous loops represents the sequential GC introduced by TinyGarble. In TinyGarble2, we provide more freedom with

support for heterogeneous loops in formats `ii` and `iii`.

$$F \equiv f(f(\ldots f()\ldots)) \tag{3.1}$$

$$F \equiv f_0(f_1(\ldots f_{Q-1}()\ldots)) \tag{3.2}$$

$$F \equiv f_P(f_0(), f_1(), \ldots, f_{Q-1}())) \tag{3.3}$$

TinyGarble2 also supports a special case of format `i`: $F \equiv f(), f(), \ldots, f()$. In this case, formally known as the *amortized* execution, the garbling of the sub-functions f are independent of each other. After every cycle, instead of transferring the shares of the output wires to the input wires, the input wires are reset to the initial states. Even though this can be performed through the traditional GC execution flow, there is a benefit in packing multiple garbling operations into one. As explained in Section 2.3, the cost of OT extension remains relatively unchanged with increasing number of OT_1^2 invocations. If multiple garbling operations are packed into one, the invocations of OT_1^2 for the input wires of all the functions are performed through one invocation of OT extension. As a result, the offline pre-processing time remains relatively unchanged, irrespective of the number of garbling, resulting in a reduction of the mean run-time. Note that a hybrid composition of these two variations is the following: $F \equiv f(f(\ldots f()\ldots)), f(f(\ldots f()\ldots)), \ldots, f(f(\ldots f()\ldots))$. Here, inputs are reset after every certain number of cycles. To enable this, we introduce the option to reset the FFs at a user-specified interval, as opposed to resetting only at cycle 0 in TinyGarble.

3.3.2 Scalability Analysis

We first analyze scalability in the honest-but-curious security model. If the number of input wires and gates in the netlist of F is $|I|$ and $|G|$, respectively, the memory footprint in traditional GC execution (i.e., all existing frameworks except [6, 29, 39, 35]) is $O\big(|I|+|G|\big)$. This is because, in the traditional execution flow, all the garbled gates and wire labels reside together in the memory. In TinyGarble2, each garbled gate is overwritten by the next one after the computation of that gate is completed. As a result, at any time instance, only one of the

garbled gates resides in the memory. However, wire labels cannot be overwritten after each use since each gate may have a fanout of more than one. The decomposition of F into smaller sub-functions creates a logical boundary beyond which the internal wires of a sub-function will not be needed. As a result, after the computation of each sub-function, the labels of the internal wires can be safely overwritten. Thus, the required memory depends on the number of input wires and the number of `AND` gates (which is also the number of internal wire labels) in a sub-function. Therefore, the memory footprint in TinyGarble2 in the honest-but-curious model is $O(|g|_M + |i|_M)$, where, $|g|_M$ is the number of gates in the largest among the netlists that constitute the system and $|i|_M$ is the largest among the numbers of inputs to the netlists.

In the honest-but-curious model, the reduction in memory footprint comes at no or negligible additional cost. However, in the malicious model, there is a trade-off between run-time and memory. The mean run-time of GC protocols in the malicious setting is significantly reduced in amortized execution. For the particular protocol, called authenticated garbling [22], adopted in this work, the mean run-time in the *function independent* phase is $O\left(\frac{|g|\rho}{log(\tau)+log|f|}\right)$, where $|g|$ is the number of `AND` gates in the function f, τ is the number of execution of f and ρ is the statistical security parameter. Note that even though the complexity is defined for amortized execution of the same function, i.e., format `i`, this particular phase is function independent, meaning that the run-time only depends on the total number of `AND` gates, irrespective of their composition or internal connections. Therefore, the run-time reduction is also available in formats `ii` and `iii` as well as functions developed with the program interface described in the next section. As a result, while decomposing the execution into smaller sub-functions results in a reduction in memory footprint, it increases the run-time.

To deal with this issue, instead of dynamically breaking up the function F into smaller sub-functions, we set the total number of `AND` gates garbled in one batch as a user-defined parameter. This allows the user to execute arbitrarily large functions at the expense of a small increase in run-time by setting the batch size according to the maximum available memory. This is not possible by the implementation of the authenticated garbling protocol by the EMP-toolkit,

for which the maximum number of gates in a function is limited by the available memory of the system. Moreover, the increase in run-time with reduced batch size diminishes with larger batch sizes. As a result, for large enough memory (few hundred MB as shown by the evaluation results in Section 3.5.2) there is no discernible change in run-time with changes in batch size.

3.4 Program Interface

The third layer of TinyGarble2 architecture – the program interface provides convenient access to the GC back-end for the users. It provides functions of common arithmetic and logical building blocks (e.g., =, +, −, ×, ÷, %, $\sqrt{}$, `if-else`, <, >, &, |, ∧ etc.) along with necessary GC primitives. Listing 1 shows the TinyGarble2 code for the *Millionaires' Problem*[1]. The list of all available operations are given in Appendix A. Currently, it supports signed integers with any bit-width from 1 to 64. Note that the GC back-end support any arbitrary length variables. To use variables with more than 64 bits from the program interface, the developers need to merge multiple integers. In the following, we describe different components of the program interface in the sequence they appear in a program.

3.4.1 Protocol Instantiation

The program starts with the instantiation of the desired GC back-end. TinyGarble2 provides two versions of the back-end for the two different security models: honest-but-curious and malicious. The rest of the interface after the protocol instantiation is identical for both models. Therefore, this is the only place where the users need to specify the security model. It is also possible to specify the security model from the command line during run-time. To the best of our knowledge, TinyGarble2 is the only framework that provides a program interface to GC execution in the malicious model and allows such a seamless switch between the two models.

[1] Yao's Millionaires' Problem is a classic example of secure two-party computation where Alice and Bob want to compare their wealth without revealing the actual wealth value.

Listing 1. TinyGarble2 code for the Millionaires' Problem

```
void main(int argc, char** argv) {
    /*set-up party (ALICE or BOB) and io
    from command line arguments*/
    TGPI = new TinyGarblePI(io, party);
    uint8_t bits = 64;
    int64_t a = 0, b = 0;
    if (party == ALICE) cin >> a;
    else cin >> b;
    tg_int a_x(ALICE, bits, a);
    tg_int b_x(BOB, bits, b);
    tg_int res_x(NONE, 1);
    TGPI->lt(res_x, a_x, b_x, bits);
    uint8_t res = TGPI->reveal(res_x, 1);
    cout << "result = " << res << endl;
}
```

3.4.2 Variables

There are three different type of *private* variables: variables owned by Alice, variables owned by Bob, and shared variables to hold the result of computations. The users need to specify the owner during variable declaration (default is shared). For the first two types of variables, the execution back-end internally generates the shares and sends them to respective parties (through OT for variables owned by Bob). This process is transparent to the user writing the program. Furthermore, the user can define any variable as a vector just by specifying the dimension during declaration. TinyGarble2 supports up to 4D vectors. This is particularly important for defining the tensors in CNN layers. The parties may choose together to reveal the actual value of any variable to either or both parties.

3.4.3 Functional Building Blocks

These building blocks are wrappers around the online computation (garbling and evaluation) of the GC protocol for the common arithmetic and logical operations. They take pre-generated shares (from either OT or a previous functional block) as inputs and generates the shares associated with the output of the function. The wrappers select the netlists according

to the operation and the bit-widths of the input. The compiled binary includes pointers to the pre-compiled netlist files from the first layer of TinyGarble2 in the installation directory. This abstraction allows the use of the most optimized netlist for a particular operation.

According to the study [33] by Mood et. al., the netlists generated by the TinyGarble framework holds less than or equal `AND` gates compared to corresponding netlists generated by the other frameworks. However, its enhanced efficiency comes from using standard logic synthesis tools and therefore needs the function to be written in a Hardware Description Language (HDL) as opposed to a programming language like `C/C++`. In TinyGarble2, netlists of the necessary operations are pre-compiled with the TinyGarble framework and provided with the software distribution. Along with the functions from [6], the first layer includes optimized versions of division and square-root operations that were first presented in [16, 17]. As a result, developers have the convenience of programming in `C++` while benefiting from the efficiency of the HDL synthesis tools. We designed a parser to convert the netlist generated by the synthesis tools to a binary file compatible with the GC back-end. This step is done only once per netlist, irrespective of the number of user-pairs or the number of GC execution per pair. In addition to the available functions, the program interface offers a *blank* wrapper that allows the users to incorporate any new Boolean netlist into their code.

Besides operations between secret variables, TinyGarble2 also supports the assignment of any secret variable to any public constant (e.g., initialization of a variable, resetting a counter) or computations involving a secret and a public variable. The complexity of addition, subtraction, comparison operations between a b-bit secret and a b-bit public value is $O(b)$, which is the same as the operation between a pair of secret values. However, for multiplication and division, the complexity becomes $O(b)$ if one of the values is public as opposed to $O(b^2)$ when two secret variables are involved.

3.4.4 Neural Network Building Blocks

As an optional fourth layer to the TinyGarble2 framework, we provide the common components for inference with Convolutional Neural Networks (CNN) - Convolution layer, Fully Connected (FC) layer, ReLU, Maxpool, ArgMax, and more. It also includes functionalities to reshape tensors containing shared variables. These building-blocks can be plugged into any CNN developed with `C/C++` by simply specifying their dimensions. Even though generally CNNs are built with Python, automated tools are available to convert any trained Python model to `C/C++` [49, 50]. The CNN building blocks in the third layer of TinyGarble2 are mostly based on the functions from its second layer. However, for three of the blocks, namely Convolution, FC, and ReLU they directly access the GC back-end.

For Convolution and FC, we provide a custom implementation of the matrix multiplication operation which forms the backbone of both these operations. Each element of the matrix product is computed through the dot products of two vectors which in turn requires a series of multiply-accumulate (MAC) operations. It is easy to compute MAC through the multiply and addition functionality of the functional layer of TinyGarble2. However, in doing so it has to alternately read the netlists for the multiplication and addition netlists for every MAC. Each layer in an NN model may include 10^5 to 10^7 MACs. Alternate reads of two netlists for such larger instances result in a small but non-negligible increase in the execution time. To tackle this issue, we compile the sequential circuit (including FFs) of the MAC netlist and garble it through the GC back-end following format `i`. ReLU operation can also be performed with the if-$else$ and comparison ($<$) function. However, it would require $2b$ `AND` gates to compute the ReLU of a b-bit variable. In our custom implementation, it only require b `AND` gates.

A unique feature of TinyGarble2, which is not available in either EMP-Toolkit or Obliv-C, is computation involving variables of mismatched bit-widths. The run-time for CNN inference through GC with b bit fixed-point representation is $O(b^2)$. The quantized model parameters of the different layers of the CNN require different numbers of bits to hold them. Moreover, in the

Convolution and FC layers, the variable to hold the result of MAC requires a larger bit-width compared to the inputs to these layers to ensure accuracy. Without the variable bit-width support, the input bit-widths need to be as large as that of MAC. This results in significant increases in run-times by EMP-Toolkit or Obliv-C, as shown by our evaluation.

3.4.5 Cautions

We conclude this section by discussing some of the facts that the developers need to be aware of. Note that most of these are inherent to the GC protocol or privacy-preserving computations in general.

Loop condition. Even though TinyGarble2 `C++` library supports the comparison operations ($<, >$), they cannot be used as conditions in `for` or `while` loops. Since none of the parties know the result of the comparison, there is no way of knowing when to end the loop. Therefore, it would result in an infinite loop.

Asymmetric operation. Both Alice and Bob have to run the same binary with the command line parameter specifying the party-ID. Moreover, every conditional operation on a secret variable, where the condition involves the party-ID, has to be symmetric. If there is a mismatch in the operations by the parties, it may result in a deadlock or undefined operations.

Operations involving the same variable. The result of operations like addition, subtraction, comparison, or logical operations between the same variable can be computed locally without going through the garbling process. If such conditions arise in the behavioral description of the function in HDL before compiling through the logic synthesis tool, that tool removes the redundant operation during optimization. However, if this condition is present in the `C++` code, our current implementation treats them as two different variables. To bypass the extra computation, we would have to check for this condition before every operation. We believe that this is a rare occasion that does not justify putting this overhead on every operation.

3.5 Evaluation of GC Frameworks

We first evaluate the performance of the Boolean logic generation of TinyGarble (which is inherited by TinyGarble2). Then we evaluate the run-time and memory usage of TinyGarble2 on two benchmarks – matrix multiplication of varying dimensions and CNN and compare them with three current frameworks, namely, EMP-Toolkit, Obliv-C, and ABY. We chose the reference frameworks based on the following criteria: (a) end-to-end framework with all the state of the art optimizations listed in Section 2.4.1, and (b) rich functionality with a standard programming language (e.g., C or C++) for developing any practical program. EMP-Toolkit, ABY and TinyGarble2 (as well as majority of the recent GC frameworks e.g., [21, 6]) sets the security parameter κ to 128. However, Obliv-C sets it to 80. Since each label is κ-bit in GC, both the run-time and memory usage is linearly dependent on its value. Therefore, while reporting the evaluation results of Obliv-C, we report the values adjusted for κ.

Evaluation Setup. We performed the experiments on an Intel Xeon CPU E5-2650 v4 @ 2.20GHz with 128GB memory running Ubuntu 18.04.3 LTS Operating System (OS). The evaluation is performed in the LAN setting with a network throughput of 500 Mbps and latency of 2 ms.

3.5.1 Synthesis

At the time of its publication, TinyGarble demonstrated superiority over the existing custom GC compilers. Even though a number of GC compilers have been developed by various research groups since then, TinyGarble still remains the most efficeint one. The Frigate [33] framework has been shown to outperform all other previous compilers, except TinyGarble. In Table 3.1, the number of non-XOR gates in selected benchmark functions generated by these two frameworks are compared.

Table 3.1. Comparison of the No. of non-XORs of TinyGarble with Frigate

Function	Frigate	TG	Improvement
Sum 1024	1,025	1,023	0.20%
Compare 1024	1026	1,023	0.29%
Hamming 160	719	159	77.89%
Mult 32	995	993	0.20%
MatrixMult 5x5 32	128,252	127,225	0.80%
AES 128	10,383	6,400	38.36%

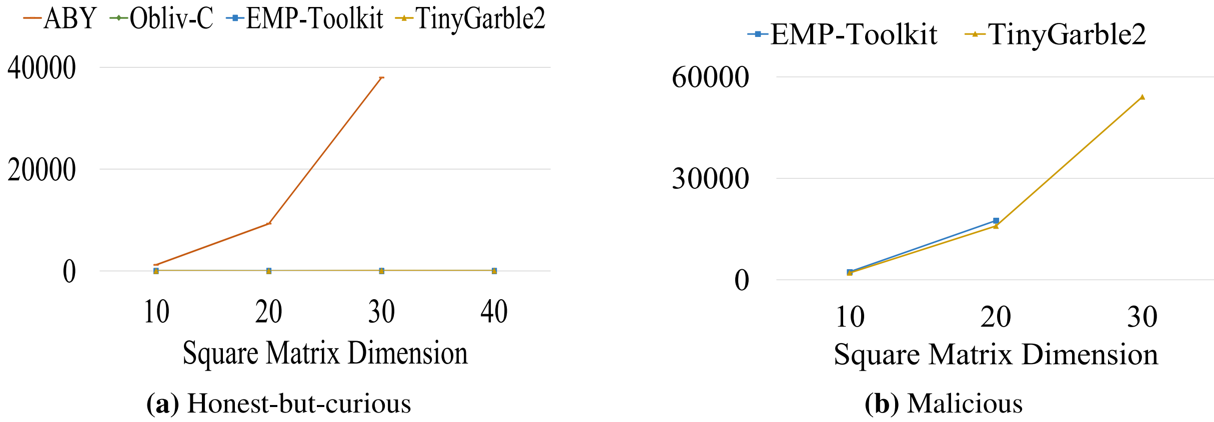
3.5.2 Runtime and Memory Footprint of Matrix-multiplication

In this experiment, we compute the product of two square matrices through GC. The bit-width of each matrix element is set to 64. First, we report the run-time by the frameworks for different dimensions of the matrices in Table 3.2. In the honest-but-curious model, TinyGarble2 performs better than both Obliv-C and ABY while having similar run-time as EMP-Toolkit. In the malicious model, supported only by TinyGarble2 and EMP-Toolkit, the run-time of TinyGarble2 is slightly longer compared to EMP-Toolkit. However, this is a small price for enabling scalability in memory footprint as we discuss next.

Table 3.2. Run-time (ms) for matrix multiplication through GC

Model	Dim.	10×10	20×20	30×30	40×40
HbC	ABY	8945	70785	335949	-
	Obliv-C	6235	24720	77038	181222
	EMP	915	4947	14689	32207
	TinyGarble2	911	5036	14948	33235
Mal	EMP	12291	96110	-	-
	TinyGarble2	13746	112003	375689	-

We plot the memory footprint for GC execution by the frameworks as a function of the matrix dimension in Figure 3.1. In the honest-but-curious model, all three frameworks, except ABY show scalability. With ABY, the memory usage by ABY increases as $O(D^3)$ for computing the product of two $D \times D$ matrices. As a result, beyond a certain maximum value of the matrix dimension ($D = 40$), the GC execution exhausts the entire available memory of the system (128GB) and is terminated by the OS. Even though EMP-Toolkit supports scalable execution in



(a) Honest-but-curious

(b) Malicious

Figure 3.1. Memory usage (MB) for matrix multiplication through GC. Batch size limitation is not applied to TinyGarble2.

the honest-but-curious model, it lacks scalability in the malicious model as shown by the plot in Figure 3.1(b). We could only compute up to $D = 20$. With TinyGarble2, we could compute up to $D = 30$, without limiting the batch-size. However, TinyGarble2 can actually compute the matrix product of any arbitrary dimension by limiting the batch size, which is shown by our next evaluation.

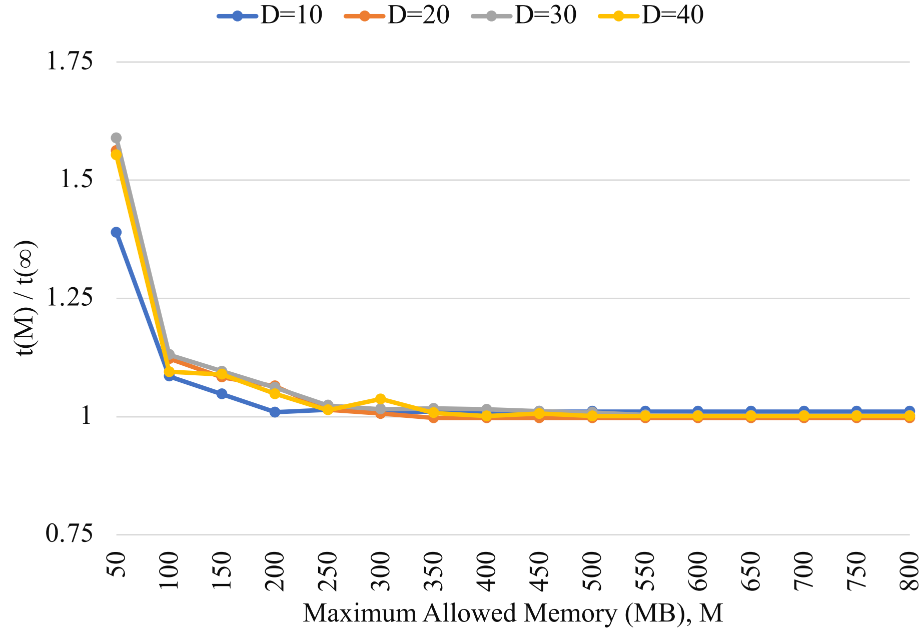


Figure 3.2. Trade-off between the run-time and memory footprint for matrix multiplication through TinyGarble2 in the malicious setting.

In this evaluation, we observe the trade-off between run-time and memory footprint for matrix multiplication through TinyGarble2. In Figure 3.2, we plot the ratio $t(M)/t(\infty)$ as a function of M, where M is the maximum allowed memory usage set by the user, $t(M)$ is the run-time for memory-limited execution, and $t(\infty)$ is the run-time for execution without memory limitation. For $D = 40$, which cannot be computed without limiting memory, $t(\infty)$ is set to $t(120\text{GB})$. The plot shows that for a small memory footprint, the run-time increases with the reduction of memory. However, with large enough memory (~ 250 MB), the run-time is almost constant.

3.5.3 Runtime and Memory Footprint of CNN Inference with LeNet-5

We have implemented the CNN model LeNet-5 [36] using the CNN library of TinyGarble2. We evaluate the model for inference on the MNIST dataset [42] with a pre-trained quantized model. In our setting, the input from Alice is the trained CNN model parameters and the input from Bob is the image. In the honest-but-curious setting, we compare the performance of TinyGarble2 with EMP-Toolkit and Obliv-C since ABY cannot handle such large operations. We also report the run-time and memory usage by TinyGarble2 in the malicious setting. None of the existing frameworks support such large operations in the malicious setting.

The bit-widths of the weights and activations vary from 16 to 24 for different layers of LeNet-5 to achieve an accuracy of 96%. However, the intermediate variables required for computation of the MACs in convolution and FC layers require more bits than the inputs to these layers. We have found through experimentation that to compute the correct results, 64 bits are required to hold the MAC outputs. Since none of EMP-Toolkit and Obliv-C support computation involving variables with mismatched bit-widths, while implementing the CNN with them we used 64-bit integers for all the inputs and intermediate variables. TinyGarble2 supports computation involving variables of any bit-width between 1 and 64, even when the bit-widths do not match. This allows us to implement a much efficient version of LeNet-5 compared to the other frameworks. In our version, we set the bit-width of the input variables of different layers to

the minimum requirements and that of the MAC outputs to 64 bits. Run-time and memory usage for inference on one image with different frameworks are presented in Table 3.3.

Table 3.3. Inference on one image with LeNet through GC

	Run-time in sec	Peak-memory in MB
Obliv-C	3.01E+03	127.73
EMP-Toolkit	3.43E+02	80.51
TinyGarble2	5.81E+01	45.83

The table shows that TinyGarble2 performs 6 ×, 18×, respectively faster and 43%, 64%, respectively more memory efficient compared to EMP-toolkit and Obliv-C. In the malicious settings, the inference time by TinyGarble2 is 74 min when the memory footprint is limited to 2GB. Even though this run-time may not be practical, the fact that TinyGarble2 can perform such large computation with only 2GB memory demonstrates its scalability in both security models.

3.5.4 Benchmarking the Program Interface

Finally, we report the run-time in the two security models for the functions available in the program interface in Table 3.4. The reported values are for 64-bit integer operations. We group similar operations with identical run-time complexity.

Table 3.4. Run-time (ms) for the operations in TinyGarble2

Operations	HbC	Mal
Add, Sub, Bit-wise AND, OR	0.02	4.48
Multiplication	0.55	12.59
Division, Modulus	1.71	45.83
Square root	2.26	30.47
Hamming	0.01	4.54

3.6 Evaluation of BMR Framework

In this section we present the evaluation result on the MPCircuits framework. We analyze two applications here – auction and voting. Later in Section 6.5, we elaborate on the Private Set

Intersection (PSI) and present the evaluation results. First, we discuss the metrics by which we characterize each MPC application. We outline the metrics and the reason for their importance in practical realization of the MPC protocols.

- Execution time (T): The total execution time of the protocol comprises the time required for garbling/evaluating the circuit (T_{GE}) as well as time spent on the communication T_C. In a general case, these two can overlap in time depending on whether the implementation is pipelined/multi-threaded or not and hence, $T \leq T_{GE} + T_C$. The distinction between the two timing parameters is important since T_{GE} mostly depends on the computational power, whereas, T_C depends on the network quality (delay and bandwidth).
- Communication ($Comm$): Maximum number of bytes exchanged between any two parties. The "maximum" is required for protocols in which communication between parties are asymmetric. In the BMR protocol, the communication between each two parties can be computed as the multiplication of number of non-XOR gates, a constant factor (=9), number of parties minus one ($n-1$), and the bit-length of each wire label (usually 128).
- Memory footprint and scalability (Mem): One of the important characteristics for each MPC protocol is the amount of memory allocated in the end-to-end execution. Protocols/frameworks that consume a high volume of memory have limited scalability in real-world scenarios where the input size from each party is large.

Evaluation Setup. The experiments are performed on a server equipped with 24 core Intel(R) Xeon(R) E5-2650 v4 @2.20GHz CPU with 256GB of RAM. We run all n parties in the same LAN network with 20ms round-trip latency and 10Gbps bandwidth.

3.6.1 Auction

In this application, each party P_i inputs a b-bit bid_i, $i = 1...n$. The outputs of computation are the index (ID) of the highest bidder i_{max} and the highest bid value $x_{pay} = max(x_1, ..., x_n)$.

We perform experiments for different numbers of participants (n) in the auction for two values of b. Table 3.5 shows the results. As can be seen, the optimized Boolean circuits using

MPCircuits technology libraries reduce the number of `AND` gates by 3.3×. Bogetoft et al. [51] have proposed a solution for secure auction based on multiple "Trusted Third Parties (TTPs)". TTPs compute the true outcome of the auction on behalf of the bidders. In this computation model, if all TTPs collude, the real input of all parties are revealed, whereas, in our approach, all parties securely process the auction and even if all other parties collude, nothing is revealed. The approach of [52] also requires a separate party called "Auction Issuer". The methodology in [43] additionally requires outsourcing the computation to two TTPs. Larson et al. [53] design a method based on a verifiable secret sharing scheme. The drawback of their approach is that not all participants in the auction are involved in the secure computation protocol and the security relies on the evaluators. Therefore, our solution is the *only* solution that (i) has constant round complexity and (ii) guarantees security even for cases where all other parties are corrupted.

3.6.2 Voting

In this application, each party P_i inputs the index of the candidate to whom she wants to vote ($vote_i$). Number of candidates is n_c and each $vote_i$ is $lg(n_c)$-bit. The outputs of computation is the index of the candidate, n_h, with the highest vote.

Table 3.6 shows the experimental results for different number of parties (voters) and candidates. As can be seen, MPCircuits is between 1.4-2.7× more efficient compared to standard utilization of logic synthesis tools. Civitas [54] is a secure voting system which is verifiable and coercion-resistant but requires five different type of agents for its execution. Fujioka et al. [55] also propose a solution for secure auctions but it requires two additional entities called administrator and the counter conspire. In contrast, our solution does not involve any additional agents or entities.

3.7 Brief Overview of Existing GC Frameworks

In this section, we present a brief overview of the existing general purpose GC frameworks. Realization of a function through GC entails two major steps: compiling the function description

Table 3.5. Evaluation on privacy-preserving auction.

		Non-optimized		**Optimized**						
b	n	#XOR	#AND	#XOR	#AND	OT (s)	T_{GE} (s)	T (s)	$Comm$ (MB)	Mem (MB)
16	4	69	324	261	97	0.74	0.62	2.39	0.04	10.25
	8	140	761	600	228	1.69	1.91	6.62	0.22	10.29
	16	281	1638	1281	492	3.51	4.48	15.06	1.01	18.14
32	4	133	660	534	194	0.74	0.66	3.41	0.08	10.31
	8	269	1547	1229	454	1.66	1.83	6.50	0.44	10.36
	16	539	3324	2621	975	3.48	4.34	16.85	2.01	30.65

Table 3.6. Evaluation on privacy-preserving voting.

		Non-optimized		**Optimized**						
n_c	n	#XOR	#AND	#XOR	#AND	OT (s)	T_{GE} (s)	T (s)	$Comm$ (MB)	Mem (MB)
2	8	7	17	18	8	1.57	1.77	9.35	3.3 KB	10.09
	16	19	43	45	16	3.29	4.29	13.76	0.02	10.09
4	4	17	50	23	37	0.71	0.54	3.25	0.02	10.09
	8	49	128	105	79	1.64	1.80	6.46	0.08	10.08
	16	123	294	249	147	2.99	4.11	14.23	0.30	10.08
8	16	250	739	545	388	3.40	4.01	15.40	0.80	15.30

to a free-XOR optimized netlist of Boolean logic and executing the GC protocol on the netlist. While some of the frameworks support both these steps, some of them focus on only one of them, more commonly the first one.

The first realization of the GC protocol is the Fairplay [43] framework. It introduces the Secure Function Definition Language (SFDL) to write the functions. The Fairplay compiler converts the functions to a netlist in Secure Hardware Definition Language (SHDL) which is later garbled through the GC execution framework.

CBMC-GC [56] presents a compiler that accepts the input in a subset of ANSI-C. It employs a bit-precise model checker, CBMC, to translate C programs into equivalent Boolean netlist in ASCII format. An updated version of this framework is presented later [34]. This framework does not include a GC-execution back-end.

The PCF framework [29] accepts the function description in a subset of C. The function is then converted to LCC byte-code through the LCC compiler and then the PCF compiler converts it to GC optimized netlist in a condensed ASCII format called Portable Circuit Format (PCF). PCF provides its own GC execution back-end to garble the netlist. One of the prominent features of its GC back-end is runtime loop unrolling. This ensures that all the garbled gates do not need to reside on the memory at the same time. However, this feature also slows down the execution. The long run-time is further aggravated by the absence of at least two of the most recent GC protocol optimizations – half gate [28] and fixed-key block cipher [21] that the framework precedes. Moreover, the generated netlists are 50-80% less optimized in terms of the number of non-XOR gates on the reported benchmarks compared to the recent frameworks.

The Obliv-C [39] framework presents an extension of the C language to develop secure applications based on GC. The main addition in Obliv-C is the `obliv` qualifier that can be applied to the C types to indicate to the compiler that the variable is private. In this framework, compilation and execution are unified into one task. The output of the framework is a compiled binary that executes the function securely through GC. ObliVM [57] is a similar framework that accepts the input function in Java and outputs a Java class file. However, this framework does not support the fixed-key block cipher optimization and therefore results in $\sim 6\times$ slower execution compared to the fastest execution back-ends. One beneficial feature available in both Obliv-C and ObliVM is the support for sub-linear oblivious access to arrays when the index depends on the private data (in general such access takes linear time w.r.t. the size of the array).

The TinyGarble [6] framework provided the most memory-efficient version of GC execution before TinyGarble2. Its introduction to sequential GC resulted in the most compact representation of any given function. However, its usability is limited by the requirement of homogeneous loops and function description in an HDL as opposed to a general-purpose programming language. The most significant contribution of TinyGarble is its ability to re-purpose existing logic synthesis tools to generate GC optimized netlist for the input function written in a Hardware Description Language (HDL) like Verilog or VHDL. To date, the circuit

generation tool-chain of TinyGarble generates the most efficient netlists.

The Frigate [33] framework accepts the input in a custom language that resembles `C`. Similar to CBMC-GC, it only provides the compiler, not the GC execution back-end. It presents an efficient representation of the netlist such that the read time is minimized during garbling. In addition to presenting the framework, the authors examine the reliability of the previous frameworks and found out that most of them suffer from reliability issues. For example, in their experiments, CBMC-GC, Obliv-C, ObliVM, and PCF crashed on programs that should have been compiled correctly. Moreover, some of the netlists generated by ObliVM, and PCF were incorrect. Many of these issues have since then been taken care of by the respective developers. Frigate and CBMC-GC are currently the most efficient among the GC compilers that accept the function in a programming language (as opposed to an HDL as in TinyGarble).

The EMP-toolkit [35] provides an efficient GC execution framework along with independent implementations of several other security primitives like OT. The most important feature of this framework is that it is one of the few publicly available tool for GC execution in the malicious setting. It implements the authenticated garbling [22] presented in Section 2.4.2. This is currently the fastest maliciously secure implementation of GC. This framework also includes a netlist generation tool. However, the efficiency of the tool is inferior compared to most of the other GC frameworks [6, 33, 56]. Moreover, even though it accepts the function description in `C++`, we found the available features to be very limited which makes it unsuitable for developing practical systems. For example, it supports comparison operation, but the result of the comparison cannot be used as an input to a subsequent operation (e.g., `obliv if` in Obliv-C, `MUX` in ABY, `if-else` in TinyGarble2). As another example, it does not support shared or secret constants. Any constant value assigned to a variable is known to either Alice or Bob or both. These limitations make the implementation of functions like ReLU or Maxpool very difficult if not impossible.

3.8 Summary

In this chapter, we presented our open-source MPC frameworks. Among them, the most recent one, inheriting the capabilities of the previous ones, is TinyGarble2 – a GC framework for developing privacy-preserving applications with `C++`. The framework presents the most optimized implementations of the basic arithmetic and logical building blocks, which can be combined to develop a wide range of practical applications. In addition to the convenience of a programming language, TinyGarble2 provides the best possible performance in terms of speed and memory efficiency. Furthermore, TinyGarble2 allows scalable execution of practical sized problems in both honest-but-curious and malicious security models – an issue not addressed by the majority of the existing GC frameworks. The enhanced capability of TinyGarble2 is demonstrated by a library for privacy-preserving CNN inference through GC. Our evaluations show that TinyGarble2 outperforms the existing frameworks both in run-time and memory usage.

Acknowledgement. This chapter, in part, has been published at (i) 2021 IEEE Security & Privacy (S&P) and appeared as: Siam U Hussain, Sadegh M Riazi, and Farinaz Koushanfar, "The Fusion of Secure Function Evaluation and Logic Synthesis", and (ii) 2020 ACM Workshop on Privacy-Preserving Machine Learning in Practice (PPMLP) and appeared as: Siam U Hussain, Baiyu Li, Farinaz Koushanfar, and Rosario Cammarota. "TinyGarble2: Smart, Efficient, and Scalable Yao's Garble Circuit", and (iii) 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) and appeared as: Sadegh M Riazi, Mojan Javaheripi, Siam U Hussain, and Farinaz Koushanfar, "MPCircuits: Optimized Circuit Generation for Secure Multi-Party Computation" (iv) 2015 IEEE Symposium on Security & Privacy (S&P) and appeared as: Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar, "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits". The dissertation author was the primary investigator of the first two papers.

Chapter 4

General Purpose Hardware Platform for Privacy-Preserving Computation

4.1 Overview

We developed two FPGA-based hardware platform to accelerate the computations required for the GC protocol. Their purpose is to facilitate the cloud servers to provide secure services to a large number of clients in parallel. In this chapter, we present our generic hardware platform FASE: FPGA Acceleration of Secure Function Evaluation [11]. As explained in Section 2.4, in GC, the underlying function is represented as a Boolean circuit, called a *netlist*. The truth-tables of that netlist is encrypted, and the computation is performed on the encrypted netlist. Generation and communication of these encrypted tables between the server and the client cause large overhead compared to the plain-text computation. For a cloud server that is communicating simultaneously to a large number of clients through parallel channels, efficient generation of the encrypted tables becomes a challenge. As we show in this paper, generating them on our FPGA accelerator brings down the protocol execution time within the practical limit.

Prior to FASE, a number of works [58, 59, 15] accelerated GC with FPGAs, including our own work MAXelerator [58]. A secure MIPS processor is presented in GarbledCPU [59], where the netlist is always the Boolean circuit of the processor, upon which the binary of the secure function is loaded. This allows the user the ease of programming in any suitable language. However, it pays the price by having to execute a large netlist. GarbledCPU provides three

versions with trade-off between speed and privacy, and even in the least secure version, the overhead is too high for practical purposes.

Our first work in this domain, MAXelerator [58] presented an FPGA accelerator for the GC execution of a Multiply-Accumulate (MAC) for matrix-multiplication, which is the basic building block of a large number of ML models. While it achieves high throughput by custom-designing this specific application, its usage is limited to a specific scenario, which prompted us to develop a general purpose accelerator. For example, in one of our case studies, when applied to the privacy-preserving recommendation system presented in [60], acceleration of the MAC operation by ~50× resulted in only 1.5× overall acceleration since only 2/3rd of the operations involved MAC. FASE, which supports any generic function, does not achieve such a high improvement on a specific operation (MAC), however, for the same problem, the overall process is accelerated by ~12×.

A generic GC accelerator on FPGA is presented in [15]. However, this design was not able to utilize the full capability of the underlying hardware for a couple of reasons. First, it employs very simple scheduling of the Boolean gates that may lead to a large number of encryption units being unused for a significant time throughout the operation. Second, it does not involve any pipeline and therefore incurs a large time gap between consecutive inputs to the encryption units. More importantly, it employs SHA-1 for encryption, which is considered not to be secure anymore [61, 62, 63]. The authors claim that it is adequate for preserving privacy in the context of garbled circuits, where cryptography is applied at many levels. However, such a statement without a formal security proof is not acceptable and may lead to security breaches.

In FASE, we employ AES [64] for encryption similar to all the recent GC realizations on either software or hardware, especially after the appearance of the fixed key block cipher optimization presented by JustGarble [21]. We also optimize the realization of the AES core specifically for GC and achieve around 17% reduction in resource usage per core compared to MAXelerator or GarbledCPU, two of the most recent secure realization of GC. Our pipelined architecture allows the encryption cores to receive one gate each cycle. To ensure the optimal

usage of the cores, i.e., minimum idle cycles, we design a scheduling algorithm built around a software simulator for our FPGA accelerator. Moreover, we design a memory management wrapper around the embedded memory to ensure optimal use of the limited read/write ports. As a result, FASE demonstrates minimum 2 orders of magnitude improvement in terms of throughput per core over [15].

4.1.1 FPGA vs GPU as Acceleration Platform

There are several advantages of an FPGA accelerator over a processor with multiple cores. In a processor, the threads communicate among themselves through shared memory resources. To ensure that the threads do not read stale variables or there are no race conditions we need to create barriers both before and after a thread accessing that memory. The time overhead of the barrier is much higher than the time of generating one garbling table. As a result, parallelizing the GC operation do not result in improvement in timing. Parallelization of garbling operation on GPU is presented in [65, 66], but these works precede the row reduction optimization described in Section 2.4.1. Therefore, they do not manage the dependency among gates. In FPGA, however, we can precisely control the operation in sync with the clock. Our FSM precisely schedules the garbling operations in the parallel cores to make sure that all the variables (in this case the labels) are written and read in order without the use of a barrier.

4.1.2 Summary of Contributions

In brief, the contributions of FASE are the following,

- We present a pipelined garbling framework that is able to receive one gate every cycle. This allows us to garble multiple gates in parallel using a single garbling core.
- We optimize the encryption core, AES, exclusively for the GC protocol. This results in 17% reduction in resource usage compared to the most recent secure FPGA realization of GC.
- We design an efficient scheduling scheme for our pipelined architecture built around a

simulator of the FPGA design. It ensures near optimal use of the encryption cores under the constraints of gate dependency and memory access collision.

- We achieve minimum 2 orders of magnitude improvement in terms of throughput per core compared to the most recent generic GC accelerator on FPGA.

The source code of FASE is available at https://github.com/siamumar/FASE.

4.2 Global Flow

4.2.1 Security Model and Terminology

In accordance with most of the recent realization of the GC protocol [6, 33, 29, 44, 34] we adopt the *honest-but-curious* security model, which assumes that both parties follow the protocol honestly yet may try to learn additional information from the information at hand. We use the term `XOR` gates to refer to `XOR`, `XNOR` and `NOT` gates, and the term non-`XOR` gates to refer to all other gates (e.g., `AND`, `OR`, `NAND`, etc). In addition to these gates, our GC framework supports D Flip-Flops (DFFs) with inputs I, and D and output Q. At reset, The value at input I passes to Q, otherwise, at each positive edge of the clock, value at input D passes to Q. The Boolean circuit representing the function F being executed through GC is referred to as the *netlist*, and the circuit that we design on FPGA to generate the garbled tables is referred to as the *circuit*. The term *netlist cycle* is used to refer to the clock cycles pertaining to the netlist, and the term *cycle* is used to refer to the clock cycles pertaining to the circuit on FPGA.

4.2.2 System Setup

The overall system setup of FASE is presented in Figure 4.1. The cloud server includes a Central Processing Unit (CPU) as the host and an FPGA-based accelerator to perform the garbling operation. The GC accelerator on FPGA generates the garbled tables along with the labels for both garbler (server) and evaluator (client) and sends them to the host CPU. The CPU stores them in a buffer (not shown in the figure) and reads back when requested for an inference

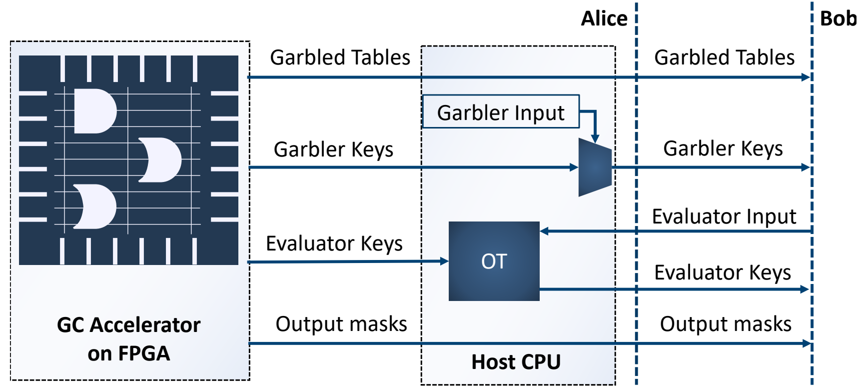


Figure 4.1. FASE system architecture on the server side.

task by a client. Note that FASE only accelerates the GC computation on the server side and is independent of the GC realization on the client side. Garbling and evaluation are similar tasks, and our garbling engine can also act as the evaluator engine with few tweaks. In general, the bottleneck of GC protocol evaluation is communication as also shown in the prior works [6]. As such, acceleration of GC evaluation on the client side is not effectual to reduce the overall latency. However, in the cloud server setting, where a single server is simultaneously communicating with a large number of clients via multiple channels, generating the garbled tables becomes the bottleneck. Therefore, accelerating this process is beneficial on the server side, but not on the client side. The presence of FASE on the server side is invisible to the clients except for the speed up in service.

4.2.3 Client-Server Model

In our setting, the cloud server acts as the garbler and the client acts as the evaluator. The motivation behind this setting is that the garbling operation does not require any input from any party. It is only during evaluation that the inputs are required. FASE keeps generating the garbled tables independently and sends them to the host CPU along with the generated labels for the input wires of the netlist. When requested by the client, the host CPU simply performs the garbling with one of the stored garbled circuits.

4.2.4 Netlist Format

The netlist is the Boolean representation of the function $c = F(a,b)$, where a, and b are inputs from the server and the client respectively. The netlist file holds information of the number of netlist input bits (i.e., the total number of bits in a and b), the numbers of FFs, XOR and non-XOR gates in the netlist, and the indices of the gates generating the final output c. It also holds information of the input and output indices, and the Boolean logic of each gate. In addition, it may also have *stall* entries indicating that the inputs of the next gate are not ready in the current cycle.

JustGarble [21] introduced the SCD format to represent the netlist. The SCD format employs efficient indexing of the gates and wires that results in a compact file. However, it requires access to multiple elements of the arrays at the same time which is not amenable to the embedded memory used to store the netlist on FPGA. FASE uses the indexing format of the SCD file but stores the netlist in a new HSCD format shown in Table 4.1 that supports reading the netlist in streaming style. D, I, INPUT_0, and INPUT_1 are indices of the inputs to the DFFs and gates respectively. LOGIC holds the 4 output bits of the gate's truth table where the inputs are in the order 00, 01, 10, 11. IS_OUTPUT is a one-bit value that is set to 1 if the DFF or gate's output is connected to the netlist output c. The index of the gate's output wire is the index of the gate in this list, thus does not need to be stored explicitly.

Table 4.1. HSCD format to store the netlist

# of Lines	Content
4	Netlist parameters (input and output bit lengths, number of dffs, gates etc)
# of dffs	D || I || 1111 || IS_OUTPUT
# of gates	INPUT_0 || INPUT_1 || LOGIC || IS_OUTPUT
# of stalls	- || - || 0000 || 0

4.2.5 Execution Steps of FASE

Our implementation is distributed over two platforms: the host CPU and FASE on the FPGA together act as the garbler. The netlist is generated at the host CPU and transferred to FASE. This step is performed only once per function, irrespective of the number of clients or the number of executions.

Then for each client, the following steps are performed.

1. FASE generates R (free-XOR, Section 2.4.1) and the AES key (fixed-key block cipher, Section 2.4.1) and sends to the host.

2. FASE generates keys for constant values 0 and 1 and sends them to the host. These keys are used if the initial values of the DFFs are assigned to constants.

3. For each netlist cycle

   (a) For each DFF

      i. If this is the first netlist cycle, the keys for the Q input of the FFs are assigned either to a constant key (corresponding to 0 or 1 depending on the value) or to the key of input I. In the latter case, the input keys are generated and sent to the host.

      ii. For the rest of the netlist cycles, keys for the Q inputs of the DFFs are copied from keys at the D inputs.

   (b) For each gate

      i. If the inputs of the gate are connected to the netlist inputs a or b, FASE generates the keys corresponding to those inputs and sends them to the host.

      ii. FASE generates the garbled table and output key and sends the garbled table to the host.

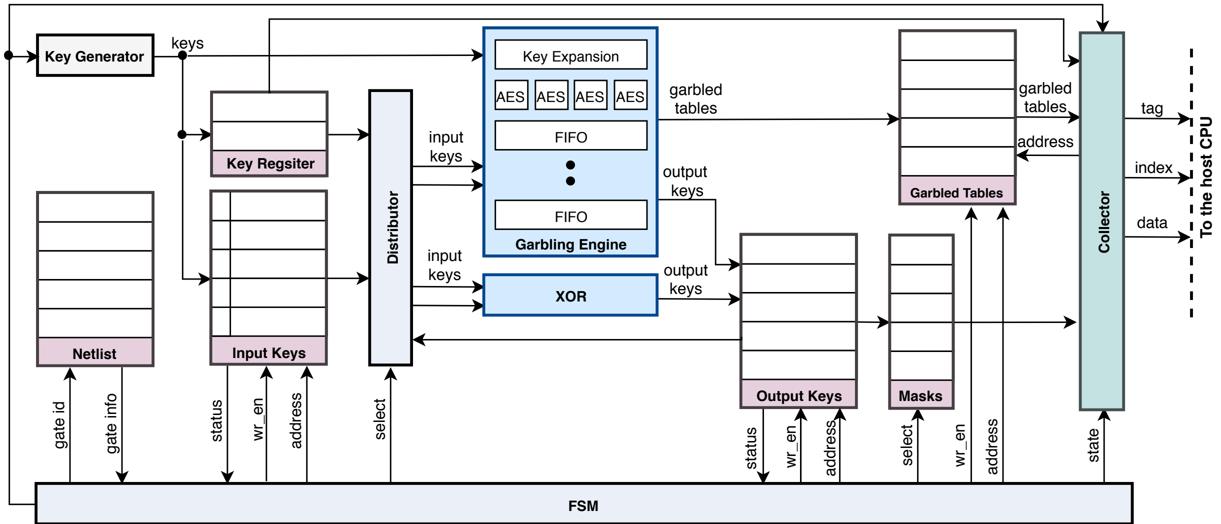


Figure 4.2. Architecture of FASE. (Please see Figure B.1 at Appendix B for an enlarged version.)

iii. If the output of the gate is connected to the netlist output y, the mask bit (Point and Permute, Section 2.4.1) is stored to an internal register file.

(c) At the end of each netlist cycle, all the mask bits are transferred to the host.

4. The host CPU performs the communication with the client, including OT, and jointly compute the output y.

Note that, generation of the garbled tables is independent of the inputs a or b. Therefore, FASE does not need any information from the host after the netlist is transferred. On the other hand, the host CPU receives the garbled tables for each non-XOR gate and the mask of each bit of the output c. However, the key of the output of each gate is only used internally inside FPGA to generate the garbled tables and outputs of the subsequent gates and not sent to the host.

4.3 Architecture of FASE

Figure 4.2 shows the different components of FASE. The heart of the system is the pipelined garbling engine that is capable of receiving one gate per cycle. Its inputs and outputs are stored in six different memories: Netlist, Key Register, Input Keys, Output Keys, Masks, and

Garbled Tables. Three of them are dedicated to storing the keys. The Key Register stores the two most recently generated keys. The Input Keys memory stores the keys associated with the netlist inputs a and b. The rest of the keys, generated either by the garbling engine or XOR is stored in the memory named Output Keys. Efficient synchronous management of these memories is key to the optimal usage of the encryption cores inside the garbling engine. The garbling operation is executed by the control logic, consisting of the Finite State Machine (FSM) and the distributor, according to the steps described in Section 4.2.5. The collector works in parallel to the control logic to collect and transfer the generated data from FASE to the host CPU. In addition, FASE incorporates a key generator for the random keys associated with the netlist inputs.

4.3.1 Key Generator

The key generator consists of 2κ True Random Number Generators (TRNG), each of which generates 1 random bit per cycle. The TRNGs are implemented with sets of ring oscillators following the design presented in [67]. The clocks to the TRNGs are controlled through two clock buffers each connected to κ TRNGs. Each set of TRNGs is only enabled when new keys need to be generated as described in Section 4.2.5.

4.3.2 Garbling Engine

Given the two keys associated with the value 0 of the two inputs and the Boolean logic of the gate, the garbling engine generates the garbled table and the key associated with the value 0 of the output. Note that according to the free-XOR optimization, the key for the value 1 of a wire is generated by XORing the key for value 0 with R. The garbling engine incorporates both the row-reduction and half-gate optimizations and thus the generated tables have two rows per gate. No garbled tables are generated for the XOR gates. Output keys of these gates are generated by the XOR block.

The garbling engine has four AES cores. They have a 10 stage pipelined architecture. Therefore, generating the garbled table and output key of each non-XOR gate requires 10 cycles.

However, the garbling engine can accept one gate per cycle due to the pipelined architecture. Even though it increases the throughput by a large margin compared to the previous accelerators, it also creates a dependency issue since the output keys of the gate, g_n sent to the garbling engine at the n-th cycle is not ready until $n+10$-th cycle. Therefore, from cycles $n+1$ to $n+9$ only the gates that are independent of the output of g_n can be sent to the garbling engine. We solve this issue by smart scheduling elaborated in Section 4.4.

As shown in Eq. 2.2, the encrypted key is again XORed with the input keys and the gate identifier. These keys are passed through 10-stage FIFOs to be XORed with the AES output.

The AES encryption function accepts two inputs: the plain-text, which is a function of the input keys, and the AES key. According to the fixed key block cipher optimization [21], the AES key is fixed for all the AES cores for one garbling session. This allows us to instantiate only one common key expansion module for all the four AES cores. As a result, the resource utilization by the garbling engine is reduced by around 17%. In our implementation, the key expansion module has 5 pipeline stages. To ensure that the expanded key is ready before the garbling of the first gate starts, we insert idle states if necessary (i.e., if the number of DFFs in the netlist is less than 4) between step (1) and (3-b) of the steps described in Section 4.2.5.

4.3.3 Control Logic

The control logic consists of the FSM and the distributor. The distributor controls the source of the input keys fed to the garbling engine. The FSM performs the following tasks:

- Fetch the current gate from the netlist and determine the source of its input keys.
- If the source is the Input Keys memory, and not already generated, turn on the key generator.
- If the source is the Output Keys memory, and not already computed, stall the operation.
- If both the input keys of the current gate is ready, increment the gate index to fetch the next gate.
- Determine the source of the output keys for each gate based on the gate logic (XOR or non-XOR).

- Control the storing of the masks based on the IS_OUTPUT value for the gate.
- Once all the garbled tables are computed, reset the gate index and increment the netlist cycle.

4.3.4 Memory Management

Netlist. This is implemented on a BRAM that stores the netlist in the HSCD format presented in Section 4.2.4. BRAMs have one cycle latency from receiving the input address (gate index) to providing the data (input indices and logic of the gate). Before proceeding to the next gate, i.e. increasing the gate index, the control logic needs to read the input indices of the current gate for checking if the input keys are ready. This would result in a latency of 2 cycles per gate. Note that the previous GC accelerators did not need to deal with this issue. MAXelerator [58] performed only one specific function and the netlist of that function was embedded into its control logic. The generic accelerator in [15] arranged the circuits in layers of independent gates and only garbled gates of one layer at one time. This approach results in FPGA resources being left unused for a large part of the operation.

Fortunately, with the indexing format of SCD [21], the gates are accessed sequentially. Therefore, the gate index is always incremented by 1. We design a wrapper around the BRAM, that always reads one address ahead of the given address (gate index) and stores the data into a register. Whenever the address is incremented, the data already stored in the register is provided at the output and the next data is requested from the BRAM. From the perspective of the control logic, this is equivalent to reading from a register file or a distributed RAM, that provides the read data immediately.

Key Register and Input Keys. The Input Keys is a dual-port BRAM to store the keys associated with the netlist inputs a and b. If the input of the current gate is connected to either of these, a new κ bit key is generated and stored in the memory, if not already generated. Otherwise, the key is read from the memory. To keep track of whether or not the key is already generated, a

register file of 1-bit *flag* registers with the same depth as the BRAM is maintained. To avoid the possibility of collisions through the read and write ports of the BRAM, the most recent pair of keys are stored in the Key Register, a register file with two κ-bit registers. There is a write to the Input Keys memory only if a new key is generated, and in that case, the keys to the garbling engine are supplied from the Key Register, eliminating the possibility of collision.

Output Keys. The keys associated with gate outputs generated by the garbling engine or XOR are stored in a dual port BRAM. These keys are also read later for subsequent gates that depend on the current gate. Unlike the Input Keys, flag registers are not required for the Output Keys since the readiness of the required keys at a certain cycle is pre-computed offline for each netlist, and encoded in the HSCD file. In [15], four cycles are required per gate for BRAM access. Reducing this time to two cycles is straight forward- using a dual port, instead of single port BRAM. However, for each gate, two keys are read from the memory and one key is written back. Therefore, theoretically, it is possible to process each gate in 1.5 cycles. To reduce the total memory access time we design a wrapper, as shown in Figure 4.3, around the BRAM.

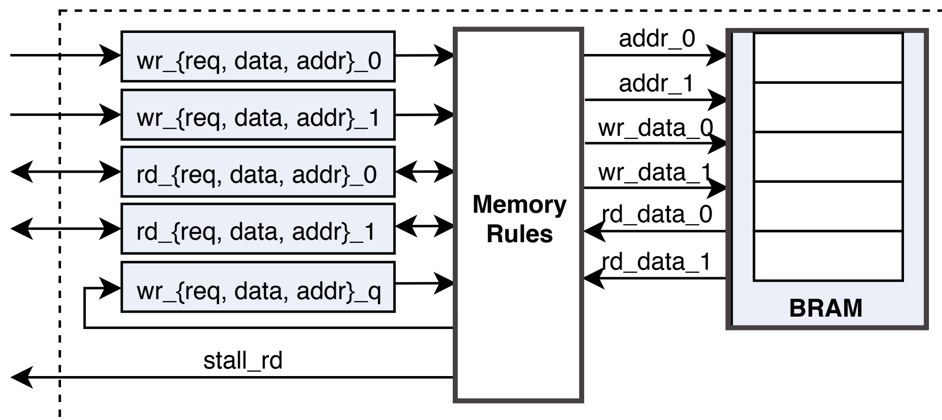


Figure 4.3. Wrapper module around the BRAM of Output Keys.

In addition to the external read and write request, address and data for the two ports 0 and 1, it has an additional output *stall_rd* that directs the control logic to stall the operation. Moreover, it has an internal queue that can hold one write command. The read and write ports of the BRAM are controlled according to the following rules.

- If there is no queued request,
  - if the number of write requests is more than or equal to the number of read requests, the write requests are performed and the read requests are stalled.
  - if the number of write requests is less than the number of read requests, the read requests are performed and the write request is queued.
- If there is a write command in the queue,
  - the read commands are stalled irrespective of the number of read requests.
  - if the number of write requests is 2, the write command at port 1 is queued and the queued write command is performed through that port.
  - if the number of write requests is 1, the queued write command is performed through the free port.

A write request is never queued for more than one cycle. Queuing of a write command is invisible to the control logic.

In addition to these, the garbled tables are stored in a dual port BRAM. The 1-bit output masks are stored in a register file so that all the mask bits can be transferred to the host CPU in one or two cycles.

4.3.5 Collector

The collector performs the communication with the host CPU. Four types of data are sent from FASE to the host: (i) the R and AES keys, (ii) keys for the netlist inputs, (iii) garbled tables for the non-`XOR` gates, and (iv) output masks for the netlist outputs. At each cycle, the collector sends the following three pieces of information to the host:

1. A *tag* indicating the type of the data being sent.
2. The *index* of the respective data.
3. The *data*.

The keys for the netlist inputs are assigned the highest priority since only the most recent pair of keys are stored in the Key Registers. The garbled tables are stored in a dual port BRAM. One of the port is used to write the garbled tables. The collector uses the other port to read them. Since the gates are accessed sequentially, the garbled tables are also written sequentially. Therefore, the read address being smaller than the write address indicates that there are new garbled tables that need to be transferred. After all the garbled tables are sent, all the output masks are sent together in one or two cycles depending on the bit length of the netlist output c.

Communication Bandwidth. Let us define the following netlist parameters. The total input bit-width is M, the output bit-width is N, the total number of gates is G, the total number of XOR gates is X, and the netlist takes C cycles to compute the operation Then the data transferred from the FPGA to the host CPU to garble one netlist is $4\kappa + ((M + 2(G - X))\kappa + N)C$ bits. This is significantly less compared to the accelerator presented in [15], which needs to transfer $(3G + 3(G - X))\kappa$ bits per netlist, the difference being $(3G + (G - X) - M)\kappa - N$. Note that [15] only supports combinational circuit, therefore the number of netlist cycles C is always 1, but the number of gates and the number of input bits will be larger. Eventually, the product $C \times G$ will be of the same order for both combinational and sequential netlists of the same function. Moreover, [15] does not support half-gate optimization, therefore the garbled table has three rows instead of two.

4.4 Scheduling the Gates

As explained in Section 4.3.2, the garbling engine is able to accept one new gate per cycle. However, since each gate takes 10 cycles to process, the gates sent to the garbling engine at cycles $n+1$ to $n+9$ should be independent of the gate g_n sent at cycle n. If not properly scheduled, there may be a large number of idle cycles, when the control logic waits for the input keys of the current gate being computed. We treat this problem as offline scheduling of a Directed Acyclic Graph (DAG) to a Bounded Number of Processors (BNP) with the number of processors set to

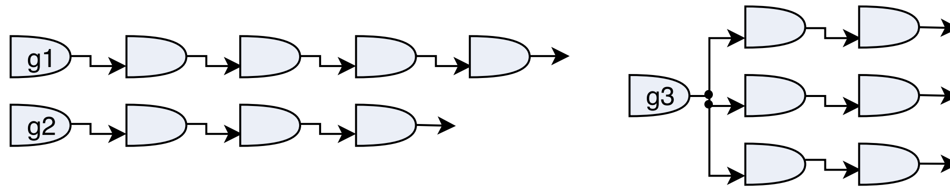


Figure 4.4. Different types of gate dependencies.

the number of pipelined stages [68, 69, 70, 71, 72]. The scheduling is performed in two steps:

1. The gates are ordered according to their priority.

2. From the ordered list, the gates are assigned one by one to one of the free processors.

4.4.1 Setting the priority

In the majority of the work on DAG scheduling, one of the three parameters are used as the measure of priority:

- t-level: length of the longest path (excluding the gate) from the netlist input to the gate.
- b-level: length of the longest path (including the gate) from the gate to netlist output.
- ALAP: length of the critical path – b-level.

The key difference between assigning tasks to parallel processors and to a single pipelined processor is that in the latter case the bottleneck is not the availability of the processors rather the readiness of the inputs. Therefore, the last two parameters are better than the t-level in this case since they both prioritize gates with a higher number of dependent gates. However, they still do not result in optimal ordering of gates. This is illustrated by a small example netlist in Figure 4.4. According to both b-level and ALAP, gates $g1$ and $g2$ have higher priority than $g3$, while scheduling $g3$ first will free up more gates. In this work, we employ the *weighted fanout* of a gate as a measure of its priority. The fanout of a gate is the number of gates dependent on it. In computing weighted fanout, the weight of `XOR` gates are set to 1, and the weights of the non-`XOR` gates are set to 10, the number of cycles it takes to compute their output keys.

4.4.2 Adding Gates to the Queue

To add gates to the queue from the ordered list we follow Algorithm 1. This is a simulator of the hardware architecture presented in Section 4.3. It includes all the constraints of the hardware (e.g., number of pipeline stages, processing one gate per cycle, memory conflict) except one. That is at every cycle, instead of reading only one gate, it reads all the gates that have not been queued yet from the ordered list and queue the first ready gate. A gate is ready when the keys assigned to its inputs have been computed. If none of the gates are ready, it inserts a $stall$.

The input to the algorithm is the ordered list of gates $GateList$ of size G. Every element of $GateList$ is a gate $g(i_0, i_1, l)$, where $i_0, i_1 \in WireList$ are the indices of the two input wires and l is the Boolean logic of g. $WireList$ is the list of wires that are ordered according to the following rules (introduced in SCD format [21]): (i) the first M indices belong to the M input wires of the netlist, and (ii) the index of the output wire of a gate is the sum of the gate's index in the $GateList$ and M. The task mem_rules at line 12 of Algorithm 1 decides if there is a stall in the read operation according to the rules outlined in Section 4.3.4.

Note that scheduling instructions in a pipelined processor is an active area of research [73, 74, 75]. However, these schemes target real-time scheduling. Therefore, they primarily optimize the speed of scheduling and deal within a limited view of different sets of operations running in parallel. In the case of GC, the gates are scheduled offline, only once per netlist, and the scheduler has the complete view of the entire netlist. Therefore, these schemes do not benefit this specific task.

4.5 Evaluation

4.5.1 Benchmark Functions

Table 4.2 shows the benchmark functions along with the number of gates and XOR gates and the number of netlist cycles to complete each function used to evaluate FASE. These benchmarks, except the MACs, are the largest sequential netlists provided in the TinyGarble [6]

Algorithm 1: Algorithm to assign gates to the queue

input : Ordered list of gates *GateList*
output : The queue of gates Q
parameters : number of input wires M
number of gates G
number of pipeline stages P

1. create arrays W_0, W_1, R_0, R_1 of 0s
2. create an array *Ready* of 0s
3. **for** $k = 1$ *to* M **do**
4.   set $Ready[k]$ to P
5. $c = 0$
6. **while** *size of* $Q < G$ **do**
7.   increment c
8.   **for** $k = 1$ *to* $M + G$ **do**
9.   **if** $Ready[k]$ *is not 0* **then**
10.   increment $Ready[k]$
11. $stall_rd =$
12. $mem_rules(W_0[c-1], W_1[c-1], R_0[c-1], R_1[c-1])$
13.   **if** $stall_rd$ *is* **true then**
14.   push $stall$ into Q
15.   **continue**
16.   **for** $k = 1$ *to* G **do**
17.   read $g(i_0, i_1, l)$ from $GateList[k]$
18.   **if** $Ready[i_0] > P$ **and** $Ready[i_1] > P$ **then**
19.   push g into Q
20.   set $R_0[c]$ to 1, set $R_1[c]$ to 1
21.   **if** l *is* XOR **then**
22.   set $W_0[c+1]$ to 1
23.   set $Ready[k]$ to P
24.   **else**
25.   set $W_1[c+P+1]$ to 1
26.   set $Ready[k]$ to 1
27.   **break**
28.   push $stall$ into Q

repository, one of the most recent and efficient netlist synthesis tools for GC. The netlists for multiplication performs the same functions as those in [6], but we use different implementations that favor parallelism. The MAC netlists perform the same function as the custom GC accelerator of MAXelerator [58].

Table 4.2. Benchmark Functions

Benchmark	Function	Input bits	# Netlist csycles	# Gates	# XORs
Mill_8_8	Millionaire's	8	8	4	3
Add_8_1	Addition	8	1	37	30
Add_8_8	Addition	8	8	5	2
Hamm_32_1	Hamming dist.	32	1	188	157
Hamm_32_32	Hamming dist.	32	32	13	8
Hamm_512_512	Hamming dist.	512	512	21	12
Mult_256_512	Multiplication	256	512	1699	1186
Mult_1024_2048	Multiplication	1024	2048	6782	4735
MAC_8_1	MAC	8	1	397	231
MAC_16_1	MAC	16	1	1678	1077
MAC_32_1	MAC	32	1	7036	4805
CORDIC_32_31	Trigonometric	32	31	2464	1544
AES_128_11	AES	128	11	4662	3225

4.5.2 Resource Utilization

FASE is implemented on a Xilinx Virtex UltraScale VCU108 (XCVU095) FPGA. The resource utilization on this platform is shown in Table 4.3. In this implementation, the number of gates $G = 2^{13}$, the number of input bits $M = 2^{10}$, and the number of output bits $N = 2^{8}$. These parameters are selected such that FASE supports the largest of the benchmark functions presented in Table 4.2. The memory requirement will increase with the increase in the values of G, M, or N. However, the resource utilization by the garbling engine and the key generator is independent of these parameters. Therefore, we report the independent utilization by the latter components separately in Table 4.3. The maximum supported clock frequency on this platform is 200MHz.

We do not compare the resource utilization with previous GC accelerators. MAXelerator [58] supports only one specific function. The accelerator in [15] employs SHA1 for

Table 4.3. Resource Utilization of FASE

Resource	Total		Garbling Engine		Key generator	
	Num	%	Num	%	Num	%
LUT	50035	9.31	31330	5.83	18202	3.39
FF	11416	1.06	5612	0.52	3917	0.36
LUTRAM	569	0.74	553	0.72	0	0.00
BRAM	68.5	3.96	0	0.00	0	0.00

encryption, which is not considered secure anymore, and 80-bit keys instead 128 bits used by the recent GC realizations. Therefore, it is not possible to make a fair comparison.

4.5.3 Evaluation of Scheduling and Memory Management

The goal of these optimizations is to reduce the number of cycles per gate. According to our evaluation, using weighed fanout instead of ALAP results in 0 to 9% reduction in the percentage of idle cycles. We choose to compare to ALAP as it has been shown to be superior over other methods of offline scheduling in a bounded number of processors [71].

To evaluate the performance improvement provided by the memory management techniques presented in Section 4.3.4, we compare the average number of cycles per gate for different benchmark functions over one netlist cycle without and with the memory management in Table 4.4. The table shows that memory management reduces the average number of cycles per gate by up to 0.5. To put these values into context, the theoretical minimum value of cycles per gate is 1.5.

4.5.4 Comparison with Previous Work

We now compare the performance of FASE with the two most recent GC accelerators implemented on FPGA [58, 15]. Both these works employ multiple cores while FASE employs a single core with a pipelined architecture. Similar to [58], we compare the performances on a per core basis. Table 4.5 compares the throughput of these works with FASE for the reported benchmark functions. Throughput is computed as the number of garbled netlist per core per cycle. FASE shows 110× to 310× improvement in throughput compared to the generic

Table 4.4. Evaluation of the Effect of the Memory Optimization

Benchmark	Without optimization		With optimization		
	# Cycles	Cycle/gate	# Cycles	Cycle/gate	Improv.
Mill_8_8	19	4.75	17	4.25	0.50
Add_8_1	120	3.24	120	3.24	0.00
Add_8_8	17	3.40	17	3.40	0.00
Hamm_32_1	342	1.82	320	1.70	0.12
Hamm_32_32	65	5.00	65	5.00	0.00
Hamm_512_512	113	5.38	113	5.38	0.00
Mult_256_512	3123	1.84	2686	1.58	0.26
Mult_1024_2048	12492	1.84	10744	1.58	0.26
MAC_8_1	769	1.94	675	1.70	0.24
MAC_16_1	3160	1.88	2770	1.65	0.23
MAC_32_1	12746	1.81	11414	1.62	0.19
CORDIC_32_31	4554	1.85	4047	1.64	0.21
AES_128_11	9004	1.93	7697	1.65	0.28

accelerator [15]. Table 4.5 also shows that the throughput of FASE is 3.65× to 4.98× smaller compared to MAXelerator [58] (presented in next chapter) for MAC operation. We would like to emphasize that [58] is a customized architecture that can only perform this one specific function while FASE is a generic GC accelerator capable of executing any given netlist.

4.5.5 Improvement in Throughput over Software Approach

Finally, we evaluate the performance of FASE against the software realization of GC presented in TinyGarble [59]. TinyGarble is built on the JustGarble [21] framework. With the fixed key block cipher optimization, this is the fastest software realization of GC at present. We run it on an Intel Xeon E5-2600 processor @ 2.9GHz with 128 GB memory. Since there is a large difference in the clock frequency of the FPGA and the CPU, we compare the performance in terms of absolute time in us, instead of the number of cycles. As shown in Table 4.6, FASE is up to 19× faster than the fastest software realization of GC.

As mentioned earlier, the customized GC accelerator for MAC presented in [58] accelerates the privacy-preserving recommendation system in [60] by only 1.5× even though it accelerates the MAC operation, which is 2/3rd of all the computations, by ~50×. In that particular work,

Table 4.5. Comparison of FASE with previous GC accelerators

Benchmark	Previous Work	# cores	# cycles	# cycles of FASE	Improv.†
Millionaire (2)	[15]	43	1.90E+02	6.70E+01	121.94
Addition (6)	[15]	43	5.60E+02	9.90E+01	243.23
Hamming (10)	[15]	43	1.20E+03	1.66E+02	310.84
Hamming (30)	[15]	43	2.20E+03	3.38E+02	279.88
Hamming (50)	[15]	43	2.80E+03	6.69E+02	179.97
A * B (8)	[15]	43	4.40E+03	6.19E+02	305.65
A * B (32)	[15]	43	3.60E+04	1.05E+04	147.78
A * B (64)	[15]	43	1.10E+05	4.28E+04	110.51
MAC_8_1	[58]	8	2.40E+01	7.01E+02	1/3.65
MAC_16_1	[58]	14	4.80E+01	2.80E+03	1/4.17
MAC_32_1	[58]	24	9.60E+01	1.15E+04	1/4.98

†In terms of (number of netlists garbled per cycle per core)

most of the remaining operations involved trigonometric functions which can be executed by the CORDIC function. FASE accelerates MAC by ~13× and CORDIC by ~10×. Therefore, it is able to accelerate the system in [60] by ~12×.

Table 4.6. Comparison of FASE on FPGA with TinyGarble [6] on CPU

Benchmark	Garbling Time(cc)	Garbling Time (μs)		
	FASE	TG [6]	FASE	Improv.
Mill_8_8	2.59E+02	1.04E+01	1.30E+00	8.05
Add_8_1	1.75E+02	8.92E+00	8.75E-01	10.19
Add_8_8	1.38E+02	1.34E+01	6.90E-01	19.37
Hamm_32_1	3.38E+02	2.98E+01	1.69E+00	17.64
Hamm_32_32	2.72E+03	1.09E+02	1.36E+01	8.00
Hamm_512_512	7.01E+04	1.98E+03	3.51E+02	5.65
Mult_256_512	1.64E+06	8.27E+04	8.19E+03	10.10
Mult_1024_2048	5.61E+07	1.25E+06	2.81E+05	4.46
MAC_8_1	7.01E+02	3.82E+01	3.51E+00	10.90
MAC_16_1	2.80E+03	1.63E+02	1.40E+01	11.62
MAC_32_1	1.15E+04	7.38E+02	5.73E+01	12.87
CORDIC_32_31	1.29E+05	6.82E+03	6.44E+02	10.60
AES_128_11	8.77E+04	5.07E+03	4.38E+02	11.57

4.6 Summary

In this chapter, we presented FASE, an FPGA accelerator for Secure Function Evaluation (SFE) by employing the Yao's GC protocol. FASE employs a pipelined garbling engine, efficient assignment of gates to the engine to reduce idle cycles, and optimized memory management to increase the number of gates garbled per cycle. As a result, it achieves at least 2 orders of magnitude improvement in throughput per core compared to the most recent generic GC accelerator on FPGA. FASE also outperforms the customized GC accelerators when applied to problems requiring diverse computations.

Acknowledgement. This chapter, in full, is a reprint of the material as it appeared at 2019 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) and appeared as: Siam U Hussain and Farinaz Koushanfar, "FASE: FPGA Acceleration of Secure Function Evaluation". The dissertation author was the primary investigator of the paper.

Chapter 5

Custom Co-design and Optimization of Privacy-Preserving Computation and Hardware

5.1 Overview

In the previous chapter, we presented a general purpose hardware platform designed to accelerate GC computation. In this chapter, we show that further speed up is possible by customizing the accelerator for specific operations. Considering the demand of efficient privacy-preserving matrix multiplication in various data-sensitive applications including Machine Learning (ML) inference, we customized our accelerator for this application.

ML models are increasingly integrated into the cloud services in order to improve the functionality of the underlying application [76, 77, 78]. The use of ML models as a cloud service has raised serious questions regarding the information privacy of clients who wants to take advantage of such services. On the one hand, clients do not want to reveal their potentially private input data (e.g., medical records, financial data, or location) to cloud servers. On the other hand, cloud servers should keep their model confidential to preserve the competitive advantage and ensure receiving continuous query requests. As such, it is highly required to devise privacy-preserving frameworks in which, ML models can be executed without disclosing the inputs of each party to one another.

In recent years, with the emergence of optimized solutions to the GC protocol, there has been increasing interest in employing GC to ensure the privacy of both the cloud and users in large-scale machine learning and data mining applications [79, 60, 80]. While significant algorithmic progress towards efficient privacy-preserving ML has been made, their usage in practical scenarios is still limited by the overhead of SFE operations. Modern ML algorithms including kernel-based data analytics [81, 82] as well as deep learning models [83] rely on iterative matrix multiplication for their execution. Matrix-based ML algorithms are key enablers for devising various contemporary data-driven applications. For instance, the well-known work by Nikolaenko et al. [60] presents a movie recommendation system with private reviews based on GC. Their system takes few hours to operate on a matrix with 10K reviews on a hardware platform with 16 cores. In this paper, we demonstrate how we can effectively reduce this computational time by around 65%. Similar to the work in [60], the bottleneck of the privacy-sensitive computation in the majority of ML applications is matrix multiplication. Real-world applications can be found in various domains such as personal finance (e.g., portfolio analysis [84]), and medical research (e.g., genome analysis [85]). For example, in portfolio analysis [84] the stock correlation data obtained by the financial institution is represented as a matrix and the stock portfolio of the client is represented as a vector and the *risk to return ratio* is the result of a multistage multiplication between these two inputs.

In this chapter, we present MAXelerator [58] – an FPGA accelerator to perform GC-based Multiply-Accumulate (MAC), which is the building block of the matrix multiplication operation. Our approach enables novel and significantly more practicable privacy preserving ML. Prior to MAXelerator, a number of works [59, 15] have presented GC implementation on FPGA. However, similar to FASE described in the previous chapter, their primary focus is on the versatility of the framework rather than computational efficiency. As explained above, in the majority of the ML computations the privacy-sensitive segment of the operation boils down to a MAC. Therefore we design a concise customized architecture on FPGA to accelerate its GC computation.

The GC architecture of MAXelerator embraces two approaches: (i) the TinyGarble [6]

framework (presented in Chapter 3) introduces sequential GC where the same netlist is garbled for multiple rounds with updated encryption keys, (ii) the work in [79] perform static analysis on the function (given the control path is independent of the data path) to determine the most optimized netlist to garble in every round. In our design, there are an outer loop and inner loops. The outer loop garbles the netlist of a MAC in every round similar to [6]. The inner loop breaks the operation of the MAC into components such that in every clock cycle we can ensure full utilization of the implemented encryption units. Unlike the conventional GC approach, the underlying netlist is embedded in a finite state machine (FSM) that controls the transfer of the keys between gates. This allows us to employ a parallel architecture for the multiplication operation as we can precisely control the garbling operation in every clock cycle and ensure accurate synchronization among the gates that are garbled in parallel. Unlike a parallelization in software, our approach does not incur any synchronization overhead. Thus we can ensure the minimal idle cycle of the encryption units. As a result, we are able to achieve respectively 57 times and 985 times improvement in throughput per core compared to [6], which is the fastest software implementation, and the FPGA implementation of [15]. Compared to FASE, MAXelerator shows $\sim 4\times$ improvement in throughput per core.

5.1.1 Summary of Contributions

The explicit contributions presented in this chapter are the following.

- Designing MAXelerator, the first hardware accelerator for privacy-preserving ML on cloud servers. The accelerator is a standalone unit that enables automated integration into reconfigurable cloud architectures.
- Presenting the first GC architecture with precise gate level control per clock cycle. Instead of conventional netlist based GC execution, we design our custom hardware accelerator as an FSM that controls the operation and communication among parallel GC cores, ensuring minimal (highest 2) idle cycles.
- Providing up to 57 times improvement in garbling operation compared to the state-of-the-art

software GC framework. This translates to the capability of the cloud to support 57 times more clients simultaneously.

- Corroborating the effectiveness of the proposed accelerator with real-world case studies in privacy-sensitive scenarios.

5.2 Global Flow

5.2.1 Security Model

Similar to FASE, MAXelerator is designed for the honest-but-curious security model, where the participating parties follow the agreed upon protocol, but may want to deduce more from the information at hand. Our hardware realization does not alter the protocol execution and thus is as secure as any software realization.

5.2.2 System Setup

We adopt a cloud server architecture with multiple channels to communicate with the clients as displayed in Figure 5.1. Along with the Central Processing Unit (CPU), the server includes MAXelerator, our FPGA-based accelerator to perform the garbling operation. The MAXelerator creates the garbled tables and sends them to the host CPU that later performs the communication with the client including OT. This setup is similar to FASE, the primary difference being the multi-core architecture as opposed the single core in FASE.

MAXelerator consists of the following components (detail description in Section 5.4):

- Parallel garbling cores to generate the garbled tables. Each core incorporates a GC engine along with a memory block to store the garbled tables.
- Label generator to create the labels necessary for the garbling operation. It includes a hardware Random Number Generator (RNG) to generate the random bit stream.
- An FSM to sync among the garbling cores. The FSM replaces the netlist in the conventional GC. This approach allows us to precisely control the garbling operation customized for

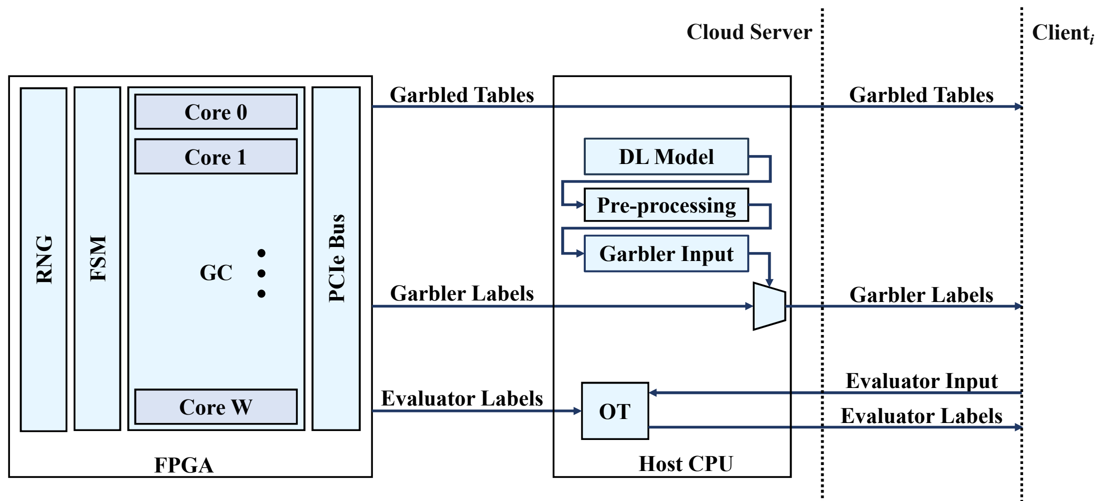


Figure 5.1. System configuration of MAXelerator framework.

the matrix-vector multiplication. Note that the netlist is embedded in the FSM. Therefore, the hardware acceleration is transparent to the evaluator (client) except for the speedup in service.

- A PCIe Bus to transfer the generated garbled tables.

5.2.3 Client-Server Model

Similar to FASE, the cloud server acts as the garbler and the client acts as the evaluator. Besides the reasoning mentioned in the previous chapter, there are few added benefits of this setting for MAXelerator. In general, the server possesses the ML model parameters (stored in the form of a set of matrices) and the client holds the input data (in the form of a single vector). Since the evaluator receives his inputs through OT, it is more efficient to have the client, who has less private data, as the evaluator. It is possible to send all the inputs at once through OT extension[23], however, the evaluator may not have enough memory to store all the labels together. With the recent development of sequential GC [6], it is feasible to perform OT every round and store only the labels required for that round; making our approach amenable to memory-constrained clients.

5.3 Architecture of MAXelerator

The control flow of the MAXelerator architecture comprises two nested loops. The product $Y_{N \times P}$ of two matrices $A_{N \times M}$ $X_{M \times P}$ is

$$Y[i,j] = \sum_{l=0}^{M-1} A[i,l]X[l,j] = \sum_{l=0}^{M-1} a[i]x[j], \quad (5.1)$$

where a and x are l-th row of A and l-th column of X, respectively. As such, the smallest unit of the matrix multiplication operation consists of a multiplier followed by an accumulator, i.e., a MAC. Following the methodology presented in [6], we design the MAC unit and garble (and evaluate) this unit sequentially for M rounds to compute one element of Y. This forms the outer loop of the control flow. Multiple parallel garbling cores are employed to generate the garbled tables. The number of cores depends on the input bit-width and available resources on the FPGA platform. In the inner loop, we breakdown the operation of the MAC unit such that (1) there is only one non-XOR operation per core per clock cycle, (2) at each cycle, no core is idle due to dependency issues. Note that the cores also contain 1 to 4 XOR gates at every cycle. However, due to the free XOR optimization they do not need costly encryption operations.

We utilize the GC optimized implementation of addition operation with the minimum number of non-XOR gates (one AND gate per input bit) provided in [6]. However, the implementation of the multiplication operation in [6] follows a serial nature that does not allow parallelism. We leverage a tree-based structure for multiplication to maximize parallelism. Figure 5.2 shows the multiplication operation of two unsigned numbers with bit-width $b = 8$. The operation for signed numbers is discussed later in this section. The bits of x (as well as their corresponding labels in GC operation) are constant over time for one multiplication, and the bits of a (as well as their corresponding labels) are input to the system serially. The addition operations represents one bit full adder where the carry is transferred internally for the next cycle. In the following, we describe the operations of the two segments marked in Figure 5.2:

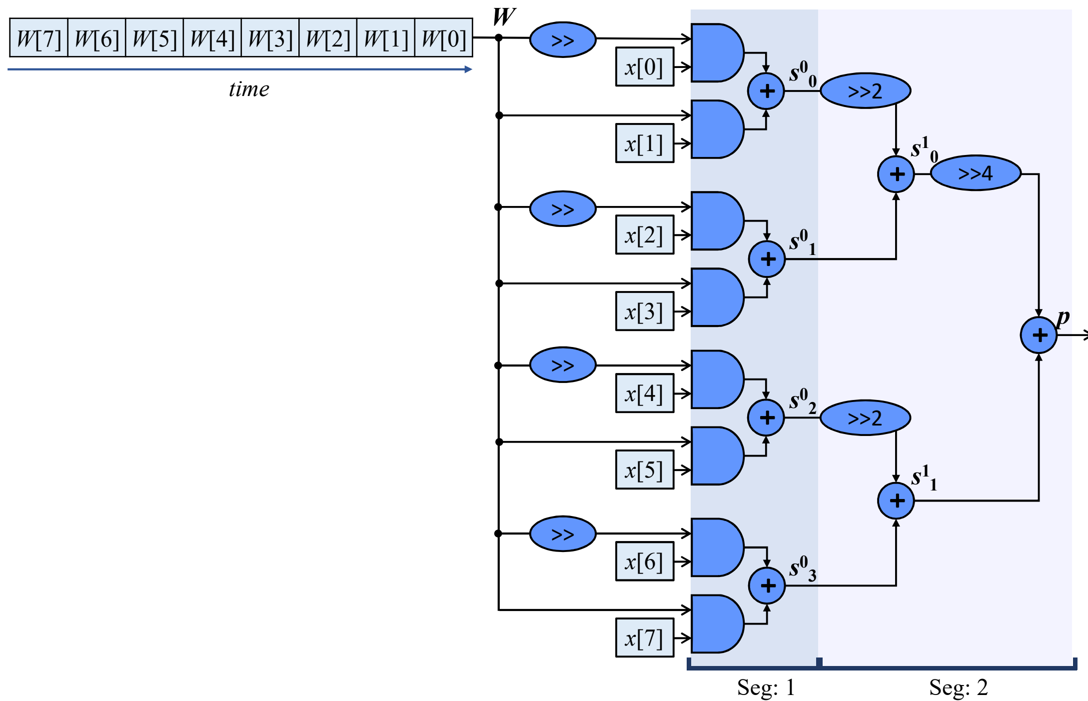


Figure 5.2. Schematic of the tree-base multiplication.

MUX_ADD and TREE.

5.3.1 Segment 1: MUX_ADD

The configuration of the parallel GC cores in segment 1 is displayed in Figure 5.3. One row in the figure represents a GC core on the FPGA, while one column represents the logic operations performed by that core in every three clock cycles. Henceforth, we refer to every three clock cycles as one *stage*. Each GC core in this segment handles two AND gates and one adder. The adder itself contains one AND and four XOR gates. The logic operations performed in one core per stage is displayed in the inset of Figure 5.3. The garbling engine of MAXelerator, as described later in Section 5.4.1, can generate one garbled table per clock cycle. Thus generating the three garbled tables requires three clock cycles, i.e., one stage.

Each core is supplied with a core id m. Core m receives the labels for the two corresponding bits of x: $x[m]$ and $x[m+1]$. These labels remain unchanged for the entire operation. All the cores then receive the labels of two bits of a: $a[n]$ and $a[n+1]$ at each stage n. However, since

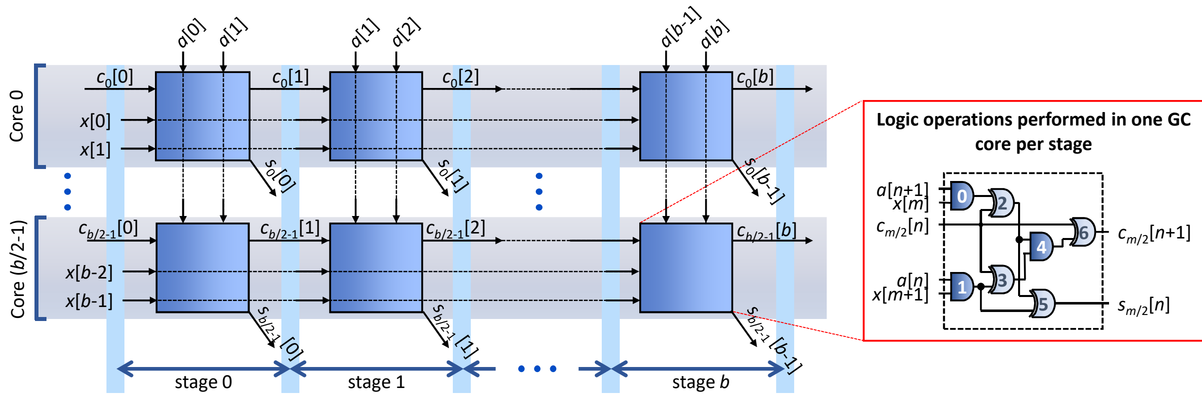


Figure 5.3. The high-level configuration and functionality of the parallel GC cores in segment 1: MUX_ADD

the garbled table for one gate is generated every clock cycle, each core needs to import only one label per cycle, thus one k-bit input port is sufficient. The label for one bit of a is required for two consecutive stages; thereby at each stage after the first, one label is ported and the other one is shifted internally.

5.3.2 Segment 2: TREE

At each stage n, a single GC core in segment 1 generates the labels for one bit of the sums: $s_0[n]$, .. $s_{b/2-1}[n]$. At the next stage, these sums are added up in segment 2 according to the tree structure. Since all the cores in segment 1 perform in parallel, the shift operations in Figure 5.2 translate to delay operations. A d stage delay is realized via d stage k-bit shift register. The number of additions performed in this segment per stage is $b/2+1$. For synchronization with segment 1, the GC cores in this segment is designed to perform three additions per core (three garbled tables, one per each addition). Thus it consists of $\lceil((b/2-1)/3)\rceil$ GC cores.

5.3.3 Accumulator and Support for Signed Inputs

The final step of the MAC is the accumulator which requires one addition per stage. To support signed inputs, two multiplexer-2's complement pairs are placed at both input and output of the multiplier. Each pair incorporates two AND gates. MAXelerator operates in a pipelined

fashion, allowing integration of these nine AND operations into segment 2. This approach results in an increased number of the shift registers. However, it ensures the minimum number of idle cycles for the GC cores. Since, the bottleneck of the resources is the number or LUTs or LUTRAMs, not the number of registers, our approach results in the most optimized design.

Performance Analysis. For bit-width b, MAXelerator requires $b/2 + \lceil (b/2+8)/3 \rceil$ cores. Thus the maximum number of idle cores is 2. The complete operation takes $b + log(b) + 2$ stages. However, since the operations are pipelined, the throughput is 1 MAC per b stages. The final throughput for the multiplication of an $M \times N$ matrix and an $N \times P$ matrix is 1 product per $MNPb$ stages or 1 product per $3MNPb$ cycles.

5.4 Hardware Setting and Results

We implement the prototype of MAXelerator on a Virtex UltraSCALE VCU108 (XCVU095) FPGA. A system with Ubuntu 16.04, 128 GB memory, and Intel Xeon E5-2600 CPU @ 2.2GHz is employed as the general purpose processor hosting the FPGA. The software realization for comparison purposes is executed on the same CPU. We leverage PCIe library provided by [86] to interconnect the host and FPGA platforms. Vivado 2017.3 is used to synthesize our GC units.

In our evaluation, we focus on comparison of MAXelerator with FPGA GC accelerators that appeared in literature before publication of MAXelerator. For comparison with our generic accelerator FASE, please see Section 4.5.4.

5.4.1 GC Engine

Each GC core incorporates one GC engine that generates one garbled table per clock cycle. The GC engine adopts all the optimizations described in Section 2.4.1. Our implementation involves only two logic gates: AND and XOR. The GC engine takes as its input the labels for the two input wires of the AND gate and outputs the output label and the two rows of the corresponding garbled tables. According to the methodology presented in [21], the encryption is

performed by fixed-key block cipher instantiated with AES. We employ four instantiations of a single stage AES implementation to perform the four required AES encryption in parallel. The s-boxes [87] inside the AES algorithm are implemented efficiently by utilizing the LUTRAMs onthe Virtex UltraSCALE FPGA. The unique gate identifier T is generated by concatenating n (see Eq. 5.1), core id, stage index and and gate id (see Figure 5.3). Due to the free XOR optimization, XOR gates just require XORing the two input labels and are handle outside while the GC engine is designed to generate garbled tables only for the AND gates. This approach ensures that there is no mismatch in the timing for executing different gates as in [59] and therefore no stalling caused by dependency issues.

The labels and garbled tables are stored in the on-chip memory of the FPGA. The memory is divided into blocks with one input port per block and one output port for the entire memory. The output port is used by the PCIe Bus to transfer the generated input labels and garbled tables to the general purpose processor hosting the FPGA. Since each core has its own block in the memory with an individual input port, logically it can be visualized as each core having its own memory block.

5.4.2 Label Generator

To generate the wire labels for GC we implement on-chip hardware Random Number Generators (RNG). We adopt the Ring Oscillator (RO) based RNG suggested in [67]. Each RO contains 3 inverters and a single RNG XORs the output of 16 ROs. The entropy of the implemented RNG on our evaluation platform is thoroughly evaluated by National Institute of Standards and Technology (NIST) battery of randomness tests [88]. In the worst-case scenario, the GC accelerator requires $k \times (b/2)$ random bits/cycle. However, for a large portion of the operation, it requires only k bits/cycle on average. The label generator incorporates $k \times (b/2)$ RNGs such that it can support the worst-case setting. The FSM that synchronizes the garbling operation fully or partially turns off the operation of the RNGs to conserve energy, when possible.

5.4.3 Resource Utilization

The FPGA resource utilization of one MAC unit is shown in Table 5.1 for different bit-widths b. It can be seen from the table that the underlying resource utilization of our design increases linearly with b. We do not compare the resource utilization with the prior-art GC implementation on FPGA [15] for two reasons: (i) [15] being a generic GC implementation, it is difficult to estimate the resource it would require only to perform the MAC operation in similar number of clock cycles as this work, (ii) it employs SHA-1 for encryption (the most resource consuming part of the implementation), while we employ AES. SHA-1 is not considered secure anymore and all the current GC realizations in both software and hardware employ AES.

Table 5.1. Resource usage of one MAC unit

Bit-width (b)	8	16	32
LUT	2.95E+04	5.91E+04	1.11E+05
LUTRAM	1.28E+02	3.84E+02	6.40E+02
Flip-Flop	2.44E+04	4.88E+04	8.40E+04

We plot the percentage of resources utilized per bit-width for one MAC unit in Figure 5.4. It can be seen from the plot that the bottleneck of the design on this platform is the number of LUTs, which reaches a peak of around 10% for $b = 32$. The maximum clock frequency supported by this implementation is 200MHz on the Virtex UltraSCALE.

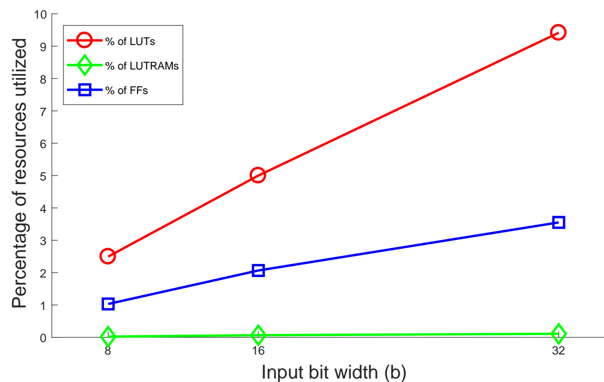


Figure 5.4. Percentage resource utilization per MAC for different bit-widths.

5.4.4 Performance Comparison with the Prior-art GC Implementation

To the best of our knowledge, MAXelerator is the first FPGA implementation of privacy-preserving deep learning. Table 5.2 compares the throughput of MAXelerator against the fastest available software GC framework TinyGarble [6] and the FPGA GC solution presented in [15]. Both MAXelerator and [15] employ parallel GC cores to accelerate the operation. In MAXelerator, the maximum number of parallel cores depends on the available resources in the FPGA while in [15] it depends on the latency of garbling one AND gate and available BRAMs on FPGA. Considering all these, we believe that reporting the overall throughput would be ambiguous and somewhat unfair to the software framework [6]. Therefore, we report the throughput of all the frameworks per core.

Table 5.2. Throughput Comparison of MAXelerator with state-of-the-art GC frameworks. Throughput is computed in number of MACs per sec

	TinyGarble [6] on CPU			FPGA Overlay Arch. [15]			MAXelerator on FPGA		
Bit-width	8	16	32	8	16†	32	8	16	32
CC per MAC	1.4E5	5.5E5	2.2E6	4.4E3	1.2E4	3.6E4	24	48	96
Time per MAC (μs)	42.29	160.35	657.65	22.00	60.00	180.00	0.12	0.24	0.48
Throughput	2.4E4	6.2E3	1.5E3	4.6E4	1.7E4	5.6E3	8.3E6	4.2E6	2.1E6
No. of cores	1	1	1	43	43	43	8	14	24
Throughput / core	2.4E4	6.2E3	1.5E3	1.1E3	3.9E2	1.3E2	1.1E6	2.9E5	8.7E4
× Throughput improve.	1/44	1/48	1/57	1/985	1/768	1/672	-	-	-

†Interpolated from the results provided in [15] for 8, 32 and 64 bits.

As shown Table 5.2, MAXelerator accelerates the garbling operation by up to 57 times compared to [6] and at least 985 times compared to [15]. Another recent GC realization on FPGA, GarbledCPU [59] do not report timing results for multiplication and addition. However, they report 2× improvement in throughput compared to JustGarbled [21] (which is the back-end of [6]) on an Intel Core i7-2600 CPU @ 3.4GHz. We estimate at least 37 times improvement over [59] in throughput per core (this work does not attempt parallelization). Due to pipeline stalls caused by dependency issues, the throughput of [6] is likely to go down further while garbling a complete netlist.

To be fair, we should state that a major factor behind the lower throughput of [15, 6] is

their focus on general purpose GC computing while MAXelerator is custom made for performing DL inference only. However, the large enhancement in throughput establishes the practicality of the custom solution.

5.5 Practical Design Experiments

In this section, we evaluate MAXelerator framework for realization of both deep learning (Section 7.1) and generic matrix-based machine learning applications (Section 7.2).

5.5.1 Deep Learning Benchmarks

We evaluate MAXelerator performance for the realization of four different DL benchmarks. Our benchmarks include both DNN and CNN models for analyzing visual, audio, and smart-sensing datasets. Table 5.3 details the computation on the server side and the transferred Bytes for each client in each benchmark. The topology of our benchmarks is outlined in the following.

Table 5.3. Number of XOR and non-XOR gates, amount of communication and computation time for each benchmark.

		#non-XOR		#XOR		Comm. (GB)		Comp. (ms)	
Id	DL Architecture	b = 8	b = 16	b = 8	b = 16	b = 8	b = 16	b = 8	b = 16
1	28×28-5C2-ReLu-100FC-ReLu-10FC-Softmax	2.0E7	6.9E7	5.5E7	1.6E8	0.68	2.25	13.04	26.07
2	28×28-300FC-ReLu-100FC-ReLu-10FC-Softmax	5.1E7	1.7E8	1.3E8	4.0E8	1.67	5.52	31.94	63.89
3	617-50FC-ReLu-26FC-Softmax	6.1E6	2.0E7	1.6E7	4.9E7	0.20	0.67	3.86	7.72
4	5625-2000FC-ReLu-500FC-ReLu-19FC-Softmax	2.3E9	7.8E9	6.2E9	1.8E0	76.90	254.22	1471.14	2942.28

Benchmark 1. Detecting objects in an image is a key enabler in devising various artificial intelligence and learning tasks. We evaluate MAXelerator practicability in analyzing MNIST dataset [89] using two different DL architectures. This data contains hand-written digits represented as 28×28 pixel grids, where each pixel is denoted by a gray level value in the range of 0-255. In this experiment, we train and use a 5-layer convolutional neural network for document classification as suggested in [90]. The five layers include: (i) a convolutional layer with a kernel of size 5×5, a stride of $(2,\ 2)$, and a map-count of 5. This layer outputs a matrix of size $5\times13\times13$. (ii) A ReLu layer as the non-linearity activation function. (iii) A fully-connected

layer that maps the $(5\times13\times13=865)$ units computed in the previous layers to a 100-dimensional vector. (iv) Another ReLu non-linearity layer, followed by (v) a final fully-connected layer of size 10 to compute the probability of each inference class.

Benchmark 2.. We train and use LeNet-300-100 as described in [36] for the MNIST dataset [89]. LeNet-300-100 is a classical feed-forward neural network consisting of three fully-connected layers interleaved with two non-linearity layers (ReLu) with total $267K$ DL parameters.

Benchmark 3. Processing audio data is an important step in devising different voice activated learning tasks that appear in mobile sensing, robotics, and autonomous applications. Our audio data collection consists of approximately 1.25 hours of speech collected by 150 speakers [91]. In this experiment, we train and use a 3-layer fully-connected DNN of size $(617\times50\times26)$ with ReLu as the non-linear activation function to analyze data within 5% inference error.

Benchmark 4. Analyzing smart-sensing data collected by embedded sensors such as accelerometers and gyroscopes is a common step in the realization of various learning tasks. In our smart-sensing data analysis, we train and use a 4-layer fully-connected DNN of size $(5625\times2000\times500\times19)$ with ReLu as the non-linear activation function to classify 19 different activities [92] within 5% inference error.

5.5.2 Generic ML Applications

Even though MAXelerator is designed to accelerate deep learning inference, it can greatly enhance the performance of many other ML applications. In this section, we analyze a number of well-known ML applications to assess the speedup provided by the custom FPGA realization of a GC based MAC operation. We assume a 32 bit fixed point system with 24 cores on MAXelerator. Note that the throughput can be increased linearly by adding more GC cores to the FPGA. For example, about 25 times more GC cores can fit in our current implementation platform.

Table 5.4. Ridge Regression Runtime Improvement

Name	n	d	Time (s) ([80])	Time (s) (Ours)	Runtime Impr.
communities11.IV	2215	20	314	7.8	39.8 ×
automobile.I	205	14	100	3.5	28.4 ×
forestFires	517	12	46	1.8	24.5 ×
winequality-red	1599	11	39	1.7	22.6 ×
autompg	398	9	21	1.1	18.7 ×
concreteStrength	1030	8	17	1.0	16.8 ×

Recommendation System. The movie recommendation system in [60] presents an efficient implementation of privacy-preserving matrix factorization which has been widely adopted by many other works such as [93, 94]. More than 2/3rd of the execution time in [60] is spent on matrix-vector multiplication for gradient computations. The complexity of the proposed matrix factorization is $O(Mlog^2M)$ where M is the total number of ratings while the complexity of the pertinent MAC operations in each operation is $O(Sd)$ where S is summation of total number of ratings and total number of movies, and d is the dimension of user/item profile. On the MovieLens dataset, each iteration of [60] takes 2.9hr. Incorporating our hardware accelerated MAC into the approach of [60] significantly reduces the gradient computation time, decreasing the total runtime per iteration from 2.9hr to 1hr (69% improvement)

Ridge Regression. This method is used to find the best-fit curve through a set of data points. The work in [80] combines both homomorphic encryption and Yao garbled circuits to efficiently perform privacy-preserving ridge regression. Their approach has $O(d^3)$ MACs, $O(d)$ square roots, and $O(d^2)$ divisions in the first phase and $O(d^2)$ MAC operations in the second phase. As such, accelerating the MAC operations would significantly improve the runtime as shown in Table 5.4 for selected datasets used in [80]. n and d are number of samples and feature size respectively.

Portfolio Analysis. To calculate the risk to return ratio based on the stock portfolio of the investor, the client stock weight vector w (which contains relative weight of stocks in the

investor's portfolio) and the financial institution stock covariance matrix cov (which is the result of financial institution's research on the market) are required. The risk to return ratio is then obtained by performing $w \times cov \times w'$ where w' is the transpose of w [84]. In [95], the authors reported 20μs to perform 252 rounds of risk to return analysis for a portfolio of size 2 on an Nvidia-k80 GPU. According to our evaluation, the same computation with privacy-preserving would take 1.33 seconds using TinyGarble and 15.23ms using MAXelerator.

In the above analysis, we assumed that the cloud server has sufficient number of communication channels and bandwidth. However, after a certain threshold, communication capability of the server may become the bottleneck of the operation. Note that MAXelerator does not affect the pertinent accuracy of the model in any of the benchmarks described above.

5.6 Summary

In this chapter, we presented MAXelerator – an efficient FPGA implementation of GC based MAC to accelerate privacy-preserving machine learning on cloud servers. MAXelerator achieves up to 57 times improvement in throughput per core compared to the fastest GC framework. Our acceleration focus is on matrix multiplication which is the most costly component in several key ML applications. Acceleration of this process can bring down the operational time in the privacy-sensitive scenario to practical limits, as verified by our case studies.

Acknowledgement. This chapter, in part, has been published at (i) 2018 ACM/IEEE Design Automation Conference (DAC) and appeared as: Siam U Hussain, Bita D Rouhani, Mohammad Ghasemzadeh, and Farinaz Koushanfar, "MAXelerator: FPGA Accelerator for Privacy Preserving Multiply-Accumulate (MAC) on Cloud Servers", and (ii) 2018 ACM Transactions on Reconfigurable Technology and Systems (TRETS) and appeared as: Bita D Rouhani, Siam U Hussain, Kristin Lauter, and Farinaz Koushanfar, "ReDCrypt: Real-Time Privacy-Preserving Deep Learning Inference in Clouds Using FPGAs". The dissertation author was the primary investigator of the first paper.

Chapter 6

Real-World Applications

6.1 Overview

Development of a framework and applications are inter dependent, especially at the initial stages of the framework development. On one hand, a framework provides abstraction from the details of the protocol execution to the application developer. On the other hand, development of large-scale practical applications helps understanding the properties required from a framework. The demands of a real-world application provides a deeper insight into the framework design, which is not available from small scale benchmark functions. In this chapter, we present four real-world privacy-preserving applications developed based on the frameworks described in Chapter 3. Table 6.1 shows the applications along with the frameworks used to develop them.

Table 6.1. Privacy-preserving applications presented in this chapter.

Applications	Frameworks
Secure localization [16, 17]	TinyGarble, MPCircuits
Authentication with noisy keys [18]	TinyGarble2
k-NNS on private data [20]	TinyGarble, MPCircuits
Private set intersection [9]	MPCircuits

While all these applications are notable in their own merit, they also played important roles in the maturing of our frameworks for privacy-preserving computation. For instance, the first publication [16] on the secure localization included a GC-based protocol, which was implemented using the TinyGarble framework. The triangulation process for localization requires

collaboration of four parties. The proposed technique based on GC (which supports two parties) was vulnerable to collusion. This prompted us to extend the capability of the synthesis libraries of TinyGarble to support BMR (which supports more than two parties), which eventually led us to develop the MPCircuits framework. The subsequent triangulation protocol [17] based on BMR have increased resiliency against collusion. As another example, authentication [18] demands the protocol to be secure in the malicious security model, whereas the GC back-end of the TinyGarble framework supports only honest-but-curious security model. For this work, we started incorporating the authenticated garbling [22] protocol which provides malicious security. This was the first seed of the TinyGarble2 framework with support for both security models. Furthermore, we enhanced the synthesis library of TinyGarble (Section 3.2.1) with new functions at different stages of development of these applications.

In the following, we present the details of the four applications with discussion on prior art, contributions to the respective fields, and evaluation results.

6.2 Secure Localization for Smart Cars

Contemporary automobiles are increasingly being equipped with advanced technologies that make significant enhancements to both functionality and safety of the vehicles. Two of the most significant improvement in this field are smart navigation system and inter-vehicle communication facilitating sharing of important information like traffic update, environmental hazards, accidents or road work. A large class of modern vehicle also includes an intra-network of processors connected to a central CPU providing Ethernet, USB, Bluetooth, and IEEE 802.11 interfaces [96]. Besides enhancing performance, these technologies also create new dimensions for attack. Thus, in addition to classic vehicular reliability requirement, security and privacy of the user should be taken into careful consideration while implanting these advanced features [96, 97, 98]. Moreover, due to the increasing reliance on these smart features, backup plans to cope with the failure of one or more components is also crucial for reliability.

We present the first privacy-preserving localization method for smart cars based on provably secure primitives. With this method, a car lost due to unavailability of GPS can send requests to three nearby cars to get assistance in finding its location. The three assisting cars then engage in a privacy-preserving triangle localization protocol to estimate the location of the lost car. The locations of all the cars including the lost car remain private.

To date, the most widely explored method to ensure user privacy in Location Based Services (LBS) is location cloaking [99, 100, 101]. In this method, instead of sending the exact location and time instant of the user, a range of area covered in a period of time is sent. To make sure that the user's location cannot be inferred from this data, the range and period are chosen such that there are at least $k-1$ other users in that area during that period, which ensures "k-anonymity" of the user. k-anonymity requires the existence of a trusted third party called anonymizer that combines the user location with locations of other users subscribed to the service. This anonymizer presents a single point to attack the system. Moreover, cloaking is also vulnerable to context-based attack and trajectory-tracing. More importantly, the approximate location results in noisy and stochastic response to the query. While this approximate response may be acceptable in some LBS scenario, for localization and navigation applications the accuracy of the method is crucial.

The work in [102, 103, 104] explored performing the location-based query (e.g., nearest neighbor) in a transformed space. These methods increase the accuracy over the cloaking approaches. However, they still have few drawbacks. For example, [102] propose three methods that either requires a semi-trusted third party or has to sacrifice accuracy or privacy for simplified operation. The authors in [102, 104] consider the privacy of only one party (client), while the data of the other party (server) is assumed to be public.

To compute accurate results while maintaining complete privacy of all the participating parties, we design two protocols employing two Secure Function Evaluation (MPC) techniques: Yao's Garbled Circuit (GC) [7] and Beaver-Micali-Rogaway (BMR) [8]. Unlike the previous methods, neither of GC or BMR protocols involve trade-off between accuracy and privacy. To

date application of MPC in LBS has been limited. The work in [105] presents application-specific solutions to some simple problems like point-inclusion, intersection, and closest pair based on GC. The work in [106] presents an implementation of the nearest neighbor query with GC. These methods require sharing encryption keys with another party, which poses a security threat. Our work is the first practical privacy-preserving location-based application that employs MPC techniques effectively and securely.

We devise two protocols where three cars assist in estimating the location of the lost car. The protocols are based on the secure computation of the triangle localization algorithm presented in [107]. In the first protocol, the three assisting cars participate in a total six invocations of the two-party GC operation such that the locations of all cars including the lost car remain private. To cope with the time constraint due to car movement, the protocol is designed such that each car can simultaneously participate in two GC operations with each of the two other cars (assuming a multi-core architecture of the processors, which is widely available at present). With this protocol, the location of the lost car is secure as long as at least one of the assisting cars does not collude with the other cars. The second protocol involves only one invocation of the multi-party BMR operation. This protocol is secure against collusion among any number of cars. However, the BMR protocol requires more computation than the GC and thus the second protocol is more time consuming than the first one.

As explained in Section 2.4, in both GC and BMR, the pertinent function is represented as a circuit consisting of Boolean logic gates (AND, OR, XOR etc). This circuits is called a *netlist*. We generate the netlists required for the localization protocol by using conventional logic synthesis tools with free-XOR optimized custom libraries following the methodologies presented in Section 3.2.1. While developing this application, we augmented the synthesis library of TinyGarble (subsequently MPCircuits) with division and square root functions, required for triangulation. We also added enhanced implementations of addition, subtraction, and multiplication to support signed inputs and overflow.

One major use case for our privacy-preserving localization is in military applications

when a lost military vehicle requires help in locating itself. It is crucial that the location of each participating vehicle remain private so that an adversarial vehicle cannot learn their location by pretending to be an ally or by tapping into the common channel. This application can also be beneficial in verifying a suspected vehicle claimed location via distance bounding with assist from three nearby cars. Generally, three verifying base stations perform distance bounding on the suspect vehicle confining it to a triangular region. However, this requires costly infrastructure which may not be available in all places. In this scenario, three other cars can act as the verifying base stations while their locations remain private and the location of the suspect vehicle is revealed only to the verifier.

6.2.1 Summary of Contributions

In brief, our contributions are as follow.

- We present the first privacy-preserving triangle localization for smart automotive systems based on provably secure primitives. We design two protocols utilizing MPC techniques such that a lost car along with three nearby cars can jointly compute the location of the lost car while the locations of all the participating cars remain private.
- We analyze the security and performance of the localization protocols in detail and demonstrate the trade-off between performance and collusion deterrence.
- We augment the circuit synthesis library of TinyGarble with functions required to generate free-XOR optimized netlists for triangle localization algorithm (square-root and division).
- Proof-of-concept implementation of our protocol demonstrates practicality of the design. The complete protocol is performed within only 355 ms.

6.2.2 Triangle Localization Algorithm

Before presenting the secure localization protocols, we review the triangle localization algorithm. Fig. 6.1 shows the setup of the algorithm provided in [107]. The car Q is lost. It requests three other cars A, B, and C to help locate itself.

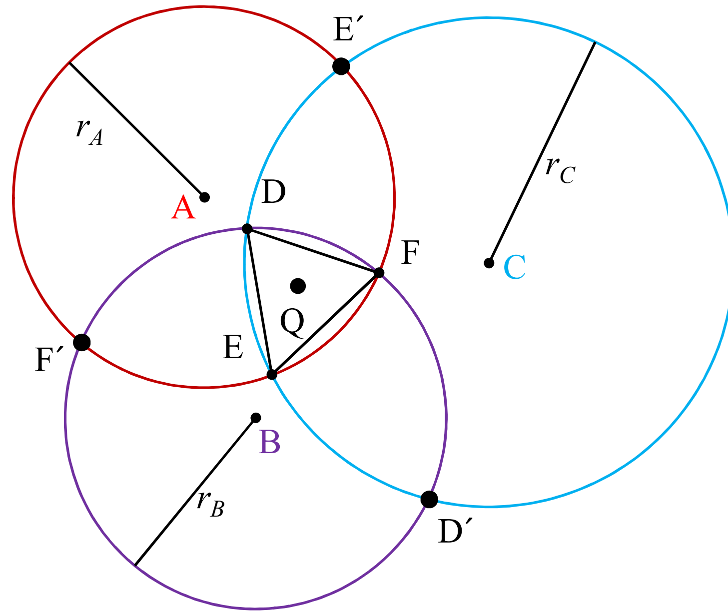


Figure 6.1. Triangle Localization Algorithm. The lost car is Q and the assisting cars are A, B, and C. The calculated location of Q is the centroid of the triangle DEF.

First, distances r_A, r_B, and r_C of Q from A, B, and C respectively, are estimated. In the ideal case where the estimated distance is exactly equal to the actual distance, the three circles centered at A, B, and C with radii r_A, r_B, and r_C, respectively, would have a common intersection at Q. However, in practice distance cannot be estimated so precisely. An underestimation may result in no intersection. Therefore, the distance is generally overestimated. In this way, a triangle DEF is formed by the points of intersections. The estimated location of Q is the median of the triangle.

In general, two circles intersect at two points (for example, circles with centers at A and B intersect at F and F'). The one that falls inside the third circle forms one vertex of the triangle (F falls inside the circle centered at C). The equations for calculating the coordinates of F and F' is provided here. The other intersections can be calculated in similar fashion. We denote the Euclidean coordinates of a point P as (x_P, y_P).

$$\sqrt{(x_F - x_A)^2 + (y_F - y_A)^2} = r_A \tag{6.1}$$

$$\sqrt{(x_F - x_B)^2 + (y_F - y_B)^2} = r_B \tag{6.2}$$

$$\sqrt{(x_F - x_C)^2 + (y_F - y_C)^2} \leqslant r_C \tag{6.3}$$

$$x_F = \frac{1}{2p}(y_F q + t) \tag{6.4}$$

$$y_F = \frac{1}{p^2+q^2}\left(pqx_A + y_B p^2 - \frac{1}{2}qt \pm \frac{1}{2}\sqrt{(qt - 2y_A p^2 - 2pqx_A)^2 - s(p^2+q^2)}\right) \tag{6.5}$$

$$\begin{aligned} where, p &= x_B - x_A,\ q = y_B - y_A \\ t &= r_A^2 - r_B^2 + x_B^2 - x_A^2 + y_B^2 - y_A^2 \\ s &= (4p^2 y_A^2 + t^2 - 4ptx_A + 4p^2 x_A^2 - 4p^2 r_A^2) \end{aligned}$$

Eq. (6.1) and (6.2) have two solutions as given by Eq.(6.4) and (6.5). The one that lies inside the range of C, decided through inequality (6.3), forms one vertex of the triangle. Note that, the vertex of the triangle is shown as F in the figure just for simplicity, it could be either of F or F'.

6.2.3 Related Work

Till present localization algorithms have been mainly used in Wireless Sensor Networks (WSN). In centroid localization, the unknown nodes location is set to the centroid of a polygon formed by the anchor nodes within a certain range. In weighted centroid localization, the centroid is calculated as the weighted mean of the coordinates of the anchor nodes [108, 109]. In triangle localization, three circles are drawn centered at three anchor nodes with the radius equal to the

estimated distances from the unknown node [110, 111, 107]. The centroid of the triangle formed by the intersection is the estimated location. In this work, we employ triangle localization as it requires only three anchor nodes while for the other techniques more anchor nodes are required for accuracy.

There are a number of works that designed privacy preserving Location Based Services (LBS) based on cryptographic primitives. Methods for privacy-preserving nearest neighbor search are presented in [102, 104]. The work in [102] employs one-way Hilbert transformation to map the space of all elements to another space and resolve the query in that transformed space. It requires a trusted third party to perform the transformation in an offline phase. The method presented in [104] confines each point of interest (POI) to a cell, named a Voronoi cell, such that the POI is the nearest neighbor to any point that falls within that cell. Then a regular rectangular grid is superimposed over this Voronoi diagram. A user retrieves all the Voronoi cells intersecting the region she belongs to on the grid through private information retrieval method and locally computes the nearest neighbor. Both these methods consider the privacy of the query only, the database of the POIs is assumed to be public. Three methods based on homomorphic encryption to find if two friends are nearby without revealing their locations is presented in [103]. There are different trade-offs involved in these methods: they either require a semi-trusted third party or sacrifice accuracy or privacy for simplified operation.

The work in [105] presents application specific solutions based on GC to several problems in location-based services. They solve basic problems like point-inclusion (whether or not one party's point is included in other party's polygon), intersection (whether or not two polygons from two users intersect), closest pair (form a pair closest to points taking one point from each set provided by two users). A GC based method to compute the nearest neighbor of a group of people is presented in [106]. In this method, two users participate in GC protocol to compute the nearest neighbor of the group. The other members of that group receive their input keys through OT from the garbler and share them with the evaluator. This creates a security threat as the collusion between only two users will reveal the location of all other members of the

group. A scalable privacy preserving k-nearest neighbor search is presented in [20] which utilize sequential description of GC [6].

6.2.4 Global Flow

The overview of the localization process is displayed in Figure 6.2. The lost car Q sends requests to three nearby cars A, B, and C to assist in computing its location. The first step is to estimate the distance r_X of Q from each assisting car X (= A, B, or C). Depending on the protocol used, either the assisting car or the lost car learns this distance, but not both of them. The location L_X of each car X is known only to itself throughout the protocol. Then A, B, C, and Q (only in the second protocol) participate in a privacy-preserving localization protocol to compute the location of Q.

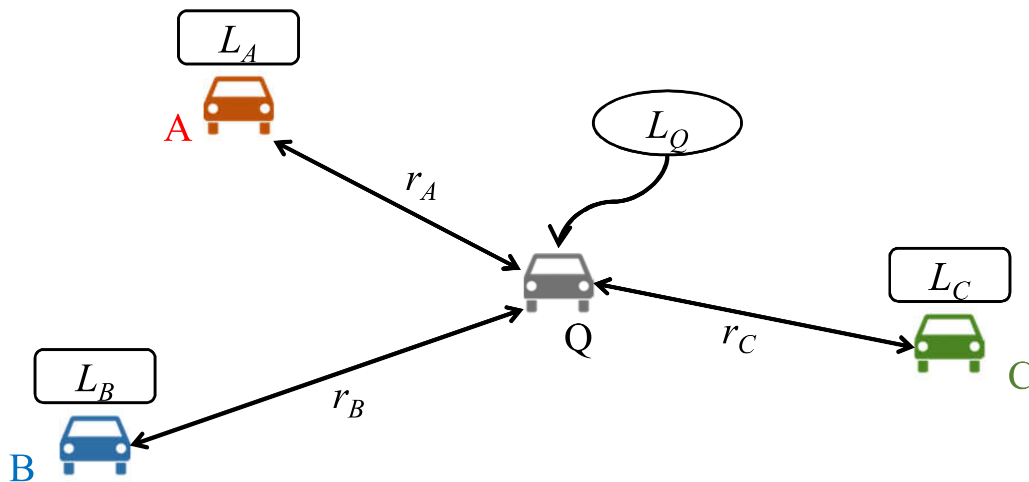


Figure 6.2. Overview of the Localization Algorithm

Ideally, the location of Q would be a common intersection of three circles centered at A, B, and C. However, due to inaccuracy in distance estimation, the location of Q is computed as the median of a triangle formed by the intersections of pairs of circles. In the first protocol, each pair of cars (say A and B) participates in a GC operation to compute two possible candidates for one vertex of the triangle. Then one of them (say B) participates in another GC operation with the third car (C) to select the candidate closer to C as the vertex. Thus, six GC operations are

required to determine all three vertices of the triangle. One car can learn zero to at most two vertices. Therefore, a single car cannot compute the median on its own. The median L_Q, i.e., the location of Q, is computed through *secure sum* [112] protocol where all four cars participate and revealed only to Q. The second protocol employs BMR, which supports more than two (in this case four) participants. In this one, the complete operation, including the computation of the median, is performed with only one invocation of the MPC protocol. Therefore, the intermediate values (intersecting points) are not revealed to any participant, making it secure against collusion.

Security Model. Consistent with the earlier relevant literature [102, 104, 103, 105, 106, 20] we adopt the *honest-but-curious* security model [29, 21]. Moreover, for privacy-preserving protocols involving more than two parties, there is the notion of *honest majority*, where the number of honest parties is higher than the number of dishonest parties. Of the two localization protocols presented in this paper, the first one requires an honest majority. However, honest majority is not a requirement for the second localization protocol.

We designed two protocols to securely compute the location of the lost car. The first one is based on the two-party MPC protocol, Yao's GC. We break down the localization function into six invocations of the GC protocol between the three assisting cars. With this protocol, the location of the lost car is secure as long as at least one of the assisting cars does not collude. The second protocol is based on the multi-party MPC protocol, BMR. This protocol is simpler and more secure than the first one as all four cars participate in one invocation of the BMR protocol. The computed location remains secure even if all three lost cars collude with one another. However, this protocol takes four times longer to compute the location as compared to the first one.

6.2.5 Protocol with Yao's GC

There are two phases in this protocol. In the first phase, the coordinates of the triangle DEF are computed through the GC protocol. For the location verification scenario, the coordinates are provided to the verifying authority after this phase. For other localization

scenarios, the median of the triangle is computed through the *Secure Sum*[112] protocol in the second phase.

Phase 1: Computing triangle DEF. For this phase, we need to evaluate the netlists of following two functions through GC. Similar to the previous section, the computation of the vertex F is used as an example here.

$[x_F, y_F, x'_F, y'_F] = Intersection(x_A, y_A, r_A, x_B, y_B, r_B)$,
that implements Eq. (6.4) and (6.5).

$in_F = Range(x_F, y_F, x_C, y_C, r_C)$,
that implements inequality (6.3).

The steps of this phase are as follows.

i Q sends LOCK_LOCATION request to A.

ii Upon receiving the request, A locks its current coordinates (x_A, y_A) and immediately start the estimation of the distance r_A with Q.

iii Steps i and ii are repeated with B and C where they lock their respective coordinates (x_B, y_B) and (x_C, y_C) immediately prior to the start of distance estimation. The estimated distances with B and C are denoted as r_B and r_C respectively.

iv A and B compute the coordinates $F(x_F, y_F)$ and $F'(x'_F, y'_F)$ of the intersections of their circles by evaluating the $Intersection$ netlist through Yao's GC protocol. The output map is configured such that A learns $F(x_F, y_F)$ and B learns $F'(x'_F, y'_F)$.

v B and C jointly decide whether F' lies inside the range of C by evaluating the $Range$ netlist through Yao's GC protocol. The output in_F is 1 if F' lies inside the range of C, and 0 otherwise, in which case the intersection F lies inside the range of C. B learns in_F and shares it with A. C learns nothing in this step.

vi B and C perform the Step iv. B learns $D(x_D, y_D)$ and C learns $D'(x'_D, y'_D)$.

vii C and A perform the Step v to compute in_D which is 1 if D' lies inside the range of A or 0 if D lies inside the range of A. C learns in_D and shares it with B. A learns nothing in this step.

viii C and A perform the Step iv. C learns $E(x_E, y_E)$ and A learns $E'(x'_E, y'_E)$.

ix A and B perform the Step v to compute in_E which is 1 if E' lies inside the range of B or 0 if E lies inside the range of B. A learns in_E and shares it with C. B learns nothing in this step.

Phase 2: Computing the median of triangle DEF. After phase 1, each assisting car possesses the coordinates of two intersections and two Boolean variables indicating whether or not these intersections are vertices of the triangle DEF. In this phase, the assisting cars along with the lost car Q compute the median of the triangle through the following steps.

i Q sends a random coordinate (x, y) to A.

ii A computes the sums $X_A = (x + \overline{in}_F.x_F + in_E.x'_E)$ and $Y_A = (y + \overline{in}_F.y_F + in_E.y'_E)$ and sends to B.

iii B computes the sums $X_B = (X_A + \overline{in}_D.x_D + in_F.x'_F)$ and $Y_B = (Y_A + \overline{in}_D.y_D + in_F.y'_F)$ and sends to C.

iv C computes the sums $X_C = (X_B + \overline{in}_E.x_E + in_D.x'_D)$ and $Y_C = (Y_B + \overline{in}_E.y_E + in_D.y'_D)$ and sends to Q.

v Q now subtracts the initial random numbers from the sums and compute the medians as $((X_C - x)/3, (Y_C - y)/3)$ which are the coordinates of its estimated location.

Security Analysis

We now analyze what information each car can learn regarding the locations of other cars.

Lost Car. In this protocol, the lost car learns nothing but its own location. However, there is a maximum range within which the cars will be able to communicate with each other. If that range is R, the lost car can assume that the three assisting cars are within a circular area around it with a radius of R. Therefore the uncertainty over the location of the assisting cars is $1/\pi R^2$.

Assisting Cars. An assisting car can be interested in two types of information: the locations of the other two assisting cars and the location of the lost car. Each assisting car knows the coordinates of only one of the intersections with the circle of the other two assisting cars. Without the coordinates of the other intersection, it is not possible to deduce the center of the other circle. Therefore, uncertainty for one assisting cars over the location of other two assisting cars is $1/\pi R^2$.

Regarding the location of the lost car, an assisting car knows the distance between the lost car and itself with some uncertainty created by the lost car by modifying the propagation time as described later in Section 6.2.8. Therefore, an assisting car X $(= A$ or B or $C)$ can confine the location of the lost car within a circular region with radius r_X. It is possible for one assisting car to know the coordinates of two of the vertices of the triangle DEF. Those two vertices form one chord of that circle. In a strict sense, it is not possible to learn which side of that chord the other vertex resides. However, if the two partitions on either side of the chord have largely different areas, it is more likely that the other vertex is on the larger partition. Even though it is not straightforward to calculate the uncertainty here, the minimum uncertainty, in this case, would be $2/\pi r_X^2$.

The regions of uncertainty for car A in locating the other cars is shown in Fig. 6.3. The uncertainty region of the lost car Q is marked with stripes and the uncertainty region of the other two assisting cars B and C is marked with dots. It is assumed that A knows the vertices E and F

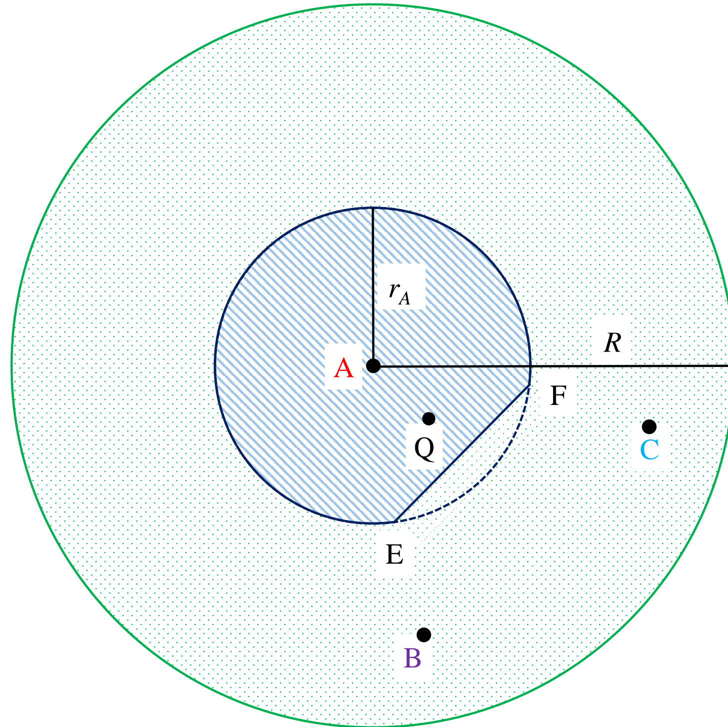


Figure 6.3. The regions of uncertainty for car A in locating the other cars. The uncertainty region of the lost car Q is marked with stripes and the uncertainty region of the other two assisting cars B and C is marked with dots.

of DEF.

Collusion Among Cars. In this protocol, the lost car Q does not participate in any invocation of the MPC protocol. Intuitively, if all three assisting cars collude with one another the location of the lost car will not remain secure. Indeed after step iii of the first phase, the cars A, B, and C collectively know all the inputs to the equations (6.3), (6.4), and (6.5). Therefore, together they can compute the location of the lost car. Another point to note here is that based on the relative location of Q, there is a possibility that one of the three assisting cars learns two vertices of the triangle while one other car knows none of them. In that case, it would be enough for two cars to collude to compute the location of the lost car. However, it is not possible to predict this scenario before the start of phase 2.

6.2.6 Protocol with BMR

The possible security breach in the previous protocol arises due to two fact that the lost car holds no inputs to the secure function. Since Yao's GC allows only two inputs, to involve the lost car in the secure computation we would have to break down both the *Intersection* and *Range* functions into two parts each and perform twelve GC operations instead of six. However, we present another protocol based on BMR that supports inputs from more than two parties.

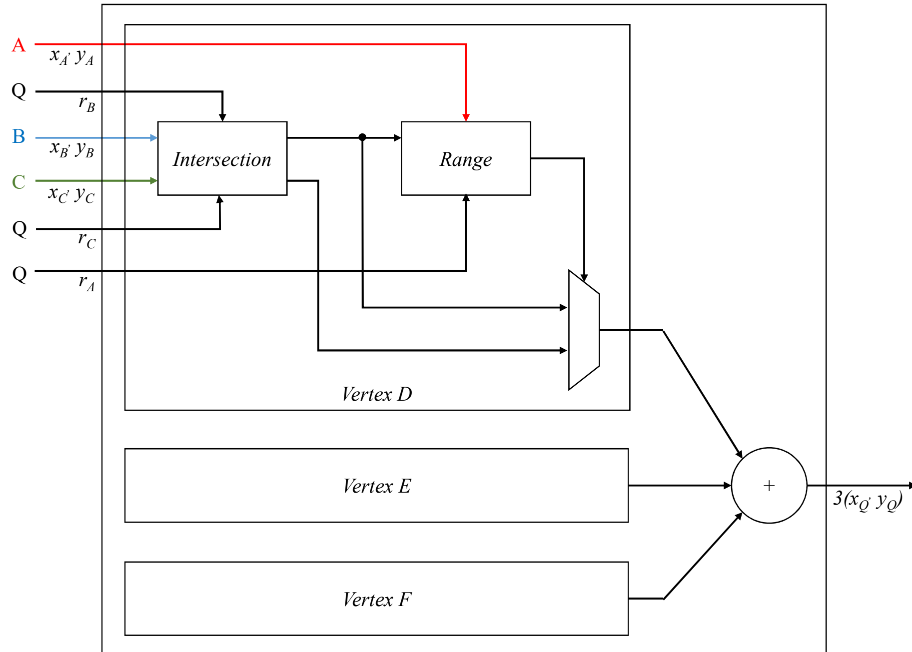


Figure 6.4. The *TriLoc* netlist to compute the location of the lost car Q with help from three assisting cars A, B, and C through the BMR protocol. Only the netlist for computing the vertex D is shown in detail.

This protocol involves only one invocation of the BMR operation where all four parties participate. It requires only one netlist which includes three instances each of the *Intersection* and *Range* netlists. The netlist, named *TriLoc*, is outlined in Figure 6.4. Only the netlist for computing vertex D is shown in detail. Unlike the first three steps of Phase 1 in the previous protocol, the distances r_A, r_B, r_C of Q respectively with A, B, and C are estimated by Q (the coordinates of A, B, and C are still locked by the respective cars). Therefore, Q now holds three inputs to the equations (6.3), (6.4), and (6.5). All of A, B, C, and Q performs garbling operation,

while only Q evaluates the netlist and thus learns the output. In location verification scenario, the output is revealed to the verifier instead of Q.

Security Analysis

The analysis on the regions of uncertainty for this protocol is similar to the first one. Since the lost car is the one estimating the distances instead of the assisting cars, their respective regions of uncertainty also switch. The lost car now can confine the three assisting cars A, B, and C within circular regions with radii r_A, r_B, r_C respectively and the assisting cars can confine the lost cars within circular regions with radius R. The regions of uncertainty of the assisting cars with respect to one another remains the same.

Collusion Among Cars. As explained above, the location of the lost car is secure with this protocol even if all three of the assisting cars collude. However, unlike the previous protocol, there is a possibility of collusion between the lost car and one or more of the assisting cars. If say C colludes with Q, then together they hold the information regarding the distances of A and C from Q: r_A and r_C, respectively. The maximum distance between A and C is $r_A + r_C$. If this distance is shorter than the maximum communication distance R, C can confine the location of A within a distance of $r_A + r_C < R$, which will result in shrinking the region of uncertainty. Since, in this protocol, the intersections between the circles are internal variables of the secure function, as shown in Figure. 6.4, the location of A cannot be predicted with an accuracy more than this.

6.2.7 Effect of the Motion of Cars

The inputs to the two functions $Intersection$ and $Range$ are locked in the first three steps of phase 1 of the protocol. The rest of the protocol execution proceeds with these locked values. Therefore, the final output of the protocol revealed to Q, is the location of Q at the end of these three steps. There are two timing constraints that affect the accuracy of the estimated location:

1. The time to lock the coordinates of A, B, and C and estimating the distances should be negligibly small such that all the cars can be considered stationary during that time

period. As shown in [113, 114] the distance estimation can be done as fast as in a few nanoseconds. Therefore the time in the first three steps primarily consists of the times to send the LOCK_LOCATION request, which is only a few bits. According to our experimentation sending a 32-bit integer takes around 1500 clock cycles which translates to around 1.5μs. Therefore the total time for these steps is around 3μs (note that the time for the LOCK_LOCATION request to the first car A does not need to be considered since the process starts only after A receives that request). Assuming the cars are moving at 100kph, they move about 83μm in this period, which is indeed negligibly small.

2. The total time of the protocol execution should be small enough so that the estimated location is close to the current location of Q. Another possibility is that Q remains stationary during the protocol execution. Note that the assisting cars do not need to be stationary since their locations are locked at the beginning. As we show in Section 6.2.13, the time to complete the protocol is 330ms. Assuming the lost car is moving at 100 kph, it will move about 9.3m during this period. Note that the current minimum accuracy of GPS coordinating systems is 8m [115].

6.2.8 Distance Compensation

According to the first protocol described in the previous section, one assisting car may know two vertices of the triangle DEF. The estimated location of Q is the median of DEF and is calculated through the secure sum protocol such that only Q learns the final result. However, if the area of the triangle is too small, the location of Q may be estimated by a car with good accuracy from just two vertices of DEF. To prevent this, Q should be allowed to manipulate the area of DEF by controlling the estimated distances from the three assisting cars. On the other hand, the estimated distance should only be known to the respective assisting car.

Among several methods available for distance estimation like RSSI (Received Signal Strength Indicator)[109, 110, 116], TOA (Time of Arrival)[117, 116, 118] , AOA (Angle of

Arrival) [119, 120] the one most suitable for this purpose is the two-way Time of Arrival method [118].

In this method, the assisting car sends a synchronization message to the lost car and the lost car sends it back after some delay. Then, the assisting car measures the time shift (t_s) between the transmitted and received messages and subtract the estimated delay t_d to get the propagation time $t_p = t_s - t_d$. In a typical application, the delay accounts for the time to receive the complete the message, and the time for the transceivers of both the cars to change their mode (transmitter $\leftrightarrow$ receiver). In this application, the lost car can wait an arbitrary time before sending back the message so that the actual delay is larger than the estimated delay t_d. This increases the estimated distance and eventually results in a larger area of DEF.

Note that since the final location is the median of the triangle, the larger area does not result in a significant error in the estimated location as we will show in Section 6.2.11.

6.2.9 Netlist Generation

We follow the TinyGarble methodology [6] to generate the netlists for GC and BMR operations. Even though TinyGarble supports both sequential and combinational circuits, the latter approach is more suited for the localization application as it does not involve repeated operation for most of the parts. The TinyGarble framework provides free-XOR optimized synthesis library that contains implementations of arithmetic functions like unsigned addition, subtraction, and multiplication. For implementations of equations (6.3) - (6.5) we extend the library by including signed versions of these functions along with support for variable bit-length and overflow, which are essential for generating the netlist for any arbitrary practical function. In addition to this, we implemented free-XOR optimized division and square-root functions as required by equations (6.4) and (6.5).

As shown in Figure 6.4, the netlist for $TriLoc$, required by the BMR based protocol, is composed of the netlists for $intersection$ and $range$ functions, along with three MUXs and one three input adder. In the following, we discuss the generation of GC/BMR optimized netlists for

these functions. The netlists for each function need to be generated only once. It is generated offline and saved in each car's memory.

Intersection. The *Intersection* netlist computes Eq. (6.4) and (6.5) that require, along with other arithmetic functions, division and square-root. In our implementation, the complexity of the number of non-XOR gates in a w-bit division operation is $O(w^2)$ which is similar to the complexity of the multiplication operation provided in [6]. The number of non-XOR gates for a 64-bit division operation is 12 546. The square root operation follows an iterative procedure. The complexity of the number of non-XOR gates in a w-bit square root operation with v iterations is $O(w^2v)$. Again, the number of required iterations can be assumed to be linearly proportional to the bit width, which simplifies the term to $O(w^3)$. Therefore, the of the number of non-XOR gates in the *Intersection* netlist with W-bit location coordinates is $O(W^3)$. The number of non-XOR gates for a 64-bit square root operation with 32 iterations is 12 733.

If we start with W-bit Euclidean coordinates, the number of bits in the internal variables keeps increasing due to overflow. The outputs of a w-bit addition/subtraction, multiplication, and division operations need $w+1$, $2w$, and w bits respectively. Going this way, inputs to the two division operations of Equation 6.5 is $3W+7$-bit (note the "±" in the equation, hence two division operations). However, the output of this equation is the Euclidean coordinates of an intersection and at the boundary condition, these coordinates can be at most four times the highest possible coordinate of an assisting car. Therefore, the outputs of these division operations will be confined to the lowest $W+2$ bits, and we can discard the rest. A similar situation occurs for the division operations for Equation 6.4. Besides reducing the number of non-XOR gates in the *Intersection* netlist, this also reduces the number of non-XOR gates in the *Range* netlist as these coordinates are its inputs.

Range. Even though inequality (6.3) involves square-root operation, both sides of this inequality are positive quantities as both of them are measured distances. Therefore, we can avoid the costly square-root operation by squaring both sides. As a result, the *Range* netlist is much

smaller than the $Intersection$ netlist, the most complex operation being squaring (multiplication) with a complexity of $O(w^2)$.

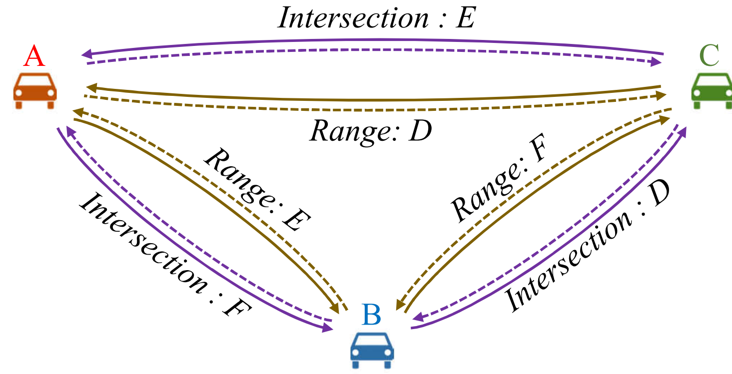


Figure 6.5. Illustration of parallel invocations of GC protocol.

6.2.10 Invocation of the MPC Protocols

GC Operation. Each of the assisting cars participates in two GC operations on the $Intersection$ netlist with the other two cars in the first protocol. These two GC operations are independent of each other and performed in parallel in two cores of the processor. To ensure symmetry, each car performs as the garbler for one pair and the evaluator for the other. Similarly, each assisting car participates in two parallel GC operations on the $Range$ netlist with the other two cars. Figure 6.5 illustrates these operations. The outer arrows depict GC on $Intersection$ and the inner arrows depict GC on $Range$. The vertex of the triangle DEF that is being computed in each GC operation is also indicated beside the arrows. A solid arrow emanating from a car indicates that the car acts as the garbler in that operation, and a dashed arrow indicates the evaluator.

The operation of the car A is described here as an example. A acts as the garbler while B acts as the evaluator to determine the coordinates of F and F' through the $Intersection$ netlist and only learns the coordinate of F. In parallel to this, A participates in another GC operation as the evaluator, with C as the garbler to compute the coordinates of E and E' and learns only the coordinate of E'. A then performs as the garbler, while B performs as the evaluator to decide

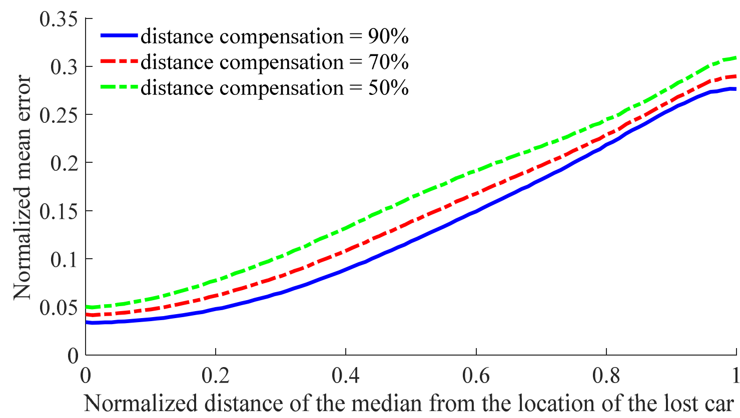
whether E' forms one vertex of the triangle through the $Range$ netlist and shares the result with C. At the same time, it acts as the evaluator in another GC operation where C is the garbler to decide whether D' forms one vertex of the triangle without learning the result.

BMR Operation. With BMR the complete operation is performed in one invocation of the protocol on the $TriLoc$ netlist. Even though the computation of each vertex is independent of each other as can be seen from Figure 6.4, BMR computes the circuit serially gate by gate. Therefore, the BMR based protocol cannot benefit from the parallelism of the operations. Moreover, as explained in Section 2.4.1, The BMR protocol incurs computation cost of $O(n^2)$ and communication cost of $O(n)$, as opposed to $O(1)$ in GC. The total number of computed gates also increases slightly since the median computation is performed through MPC instead of the secure sum as in the GC based protocol. As a result, while this protocol shows better resilience against collusion, it is slower than the first one. All of A, B, C, and Q act as garblers while only Q acts as the evaluator and learns the final output which is its location. Unlike the GC based protocol, the intermediate results, i.e., the coordinates of the intersections are not revealed to any car.

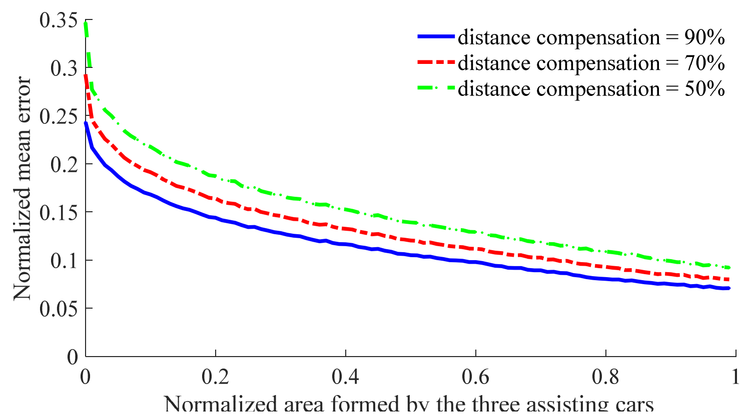
6.2.11 Evaluation: Error Analysis

We first analyze the error in the location estimated by triangle localization algorithm. Note that this error is solely due to the localization method, and distance estimation error. The MPC protocols do not introduce any additional error. To estimate the error, we run simulation by placing the assisting cars at random positions inside a square area with dimension T and place the lost car at the center of that square. The error is quantified as the Euclidean distance between the estimated and actual location of the lost car, normalized to T. The estimation error depends on two factors: (a) the relative positions of the assisting cars with respect to the lost car, (b) the area of the triangle formed by the three assisting cars.

In Figure 6.6a the error is plotted against the distance (normalized to T) between the actual location of the lost car and the median of the triangle formed by cars A, B, and C. For each



(a) Normalized mean error in the estimated location of the lost car as a function of the normalized distance between the actual location of the lost car and the median of the triangle ABC with different degrees of distance compensation.



(b) Normalized mean error in the estimated location of the lost car as a function of the normalized area of the triangle ABC with different degrees of distance compensation.

Figure 6.6. Error Analysis.

point on the curves, the simulation is run for $5.7E+03$ times. The plot shows that the estimation error increases linearly with the relative distance between the location of the lost car and the triangle ABC. To analyze the effect of distance compensation, we simulate three cases where the actual distance is increased by 50%, 70%, and 90%, respectively. The plot shows that the estimation errors are fairly close for all three cases.

In Figure 6.6b the error is plotted against the area (normalized to T^2) of the triangle formed by cars A, B, and C. For each point on the curves, the simulation is run for $2E+4$ times. The plot shows that the estimation error is high when the area is small, i.e, when the three assisting cars lie close to a straight line. The error decreases sharply with increase in the area. Similar to the previous case, distance compensation does not have a significant effect on the estimation error.

In cases where there are more than three assisting cars are available, it would be beneficial to choose the set of three cars that will result in the highest accuracy. Choosing the set according to the relative location of the assisting cars with respect to the lost car is not feasible since it requires the knowledge about the location of the lost car. However, it is possible to compute the area formed by three cars and compare it against a predetermined threshold. To ensure privacy this computation is performed by the BMR protocol.

6.2.12 Evaluation: Circuit Synthesis

As explained in Section 2.4.1, to compute a function securely through the Yao's GC or the BMR protocol, the function needs to be represented as a netlist of Boolean logic gates. Three netlists are required for the MPC operations- *Intersection* and *Range* for GC and *TriLoc* for BMR. The equations for the first two netlists (Eqs.(6.4), (6.5), and(6.3)) are described using Verilog HDL and compiled with the Synopsys Design Compiler [121] with our custom libraries. The *TriLoc* netlist is constructed from the first two. Due to the free-XOR optimization, the `XOR` gates can be computed locally without costly cryptographic encryption or communication. Therefore, the total time to compute the function is determined solely by the number of non-`XOR` gates in

the netlist. The number of non-XOR and XOR gates in the three netlists are presented in Table 6.2. It shows that the number of non-XOR gates are around only one-quarter of the total number of gates. This demonstrates the effectiveness of our customized synthesis library in generating the MPC-optimized netlist.

Table 6.2. Number of XOR and non-XOR gates in the netlists

Netlist	No. of non-XOR gates	No. of XOR gates	Total no. of gates
Intersection	2.40E+04	6.71E+04	9.11E+04
Range	4.51E+02	7.54E+02	1.21E+03
TriLoc	7.38E+04	2.06E+05	2.80E+05

6.2.13 Evaluation: Timing

To assess the timing performance, we run the two localization protocols on a system with Ubuntu 14.10 Desktop, 12.0 GB of memory, and Intel Core i7-2600 CPU @ 3.4GHz. We employ the TinyGarble framework [6] to perform the GC operations. The number of clock cycles in every phase of the GC operation to garble/evaluate the *Intersection* and *Range* netlists once is presented in Table 6.3. In the first localization protocol, each of these netlists is garbled/evaluated three times by the three assisting cars in parallel. The total number of clock cycles from the lost car initiating the operation to the final computation of its location is $1.20E+09$ which translates to only 355 ms. However, as described in Section 2.4.1, the input values to the functions are not required during the garbling operation. They are only required at the start of the oblivious transfer phase. Thereof ore, one way to reduce the accuracy loss due to the movement of the lost car is to lock the coordinates of the assisting cars after the garbling is done.

To run the BMR based protocol, we employ the framework provided at [122, 46]. Unlike GC, each car acts as garbler and only the lost car Q acts as the evaluator. The average number of clock cycles at different stages of the BMR protocol with the $TriLoc$ netlist is presented in Table 6.3. The complete protocol execution takes $8.97E+09$ clock cycles which translates to 2646 ms. As expected, the BMR based protocol have a longer run time than the GC based protocol.

Similar to the previous case, the assisting cars may wait till the end of the garbling phase before locking their coordinates. Note that in both cases the protocol execution will have to wait till all three assisting cars join. That wait time is not included in this evaluation.

Table 6.3. Timing results

Function	Garbling	Oblivious Transfer		Communication		Evaluation
		Garbler	Evaluator	Garbler	Evaluator	
Intersection	2.97E+07	3.18E+08	2.94E+08	6.06E+05	3.16E+07	2.34E+07
Range	3.65E+05	3.06E+08	2.83E+08	5.40E+04	3.55E+05	2.96E+05
TrilLoc	8.90E+08	6.53E+09	7.31E+09	7.25E+09	7.25E+09	1.36E+08

Even though the evaluation is performed on a desktop PC, this protocol is practical with processors available in smart cars today. For example, Intel Atom Processor E3845, designed for in-vehicle solutions, has four cores operating at 1.91GHz and an L2 cache of 2MB [123]. The protocol requires transmission of about 1MB of data. With transmission speed in MHz range [124], the transmission time is within practical limits. The memory footprint of this operation is about 1.8MB, which can fit in the L2 cache of an Atom processor.

6.3 Authentication with Noisy Keys

With the recent rapid surge of the Internet of Things (IoT) paradigm, remote authentication of devices holding sensitive information has become one of the primary security concerns. Traditional methods of authentication using secret keys stored in a non-volatile memory have been shown to be vulnerable to physical attacks [125, 126]. Therefore, Physical Unclonable Functions (PUF) [19] has become a key element in remote authentication of resource-constrained devices. PUFs offer a secure and lightweight alternative where the secret is not stored rather extracted from the unique physical properties of the authenticating device. Since PUFs extract the key from hardware imperfection, it is not possible to exactly reproduce these keys. Therefore, in contrast to conventional authentication, in the PUF-based methods a user/device is authenticated if the submitted key is within a certain pre-specified distance from the stored key. This requires

authentication schemes designed for *noisy* keys.

A number of PUF based authentication schemes have been presented in literature [127, 128, 129, 130, 131, 132]. However, all these schemes involve key updates and therefore require *strong* PUFs, which have an exponential number of challenge-response pairs (CRP) as opposed to *weak* PUFs, which have a limited number of CRPs. Strong PUFs, in general, require dedicated hardware and may not be suitable for resource-constrained devices which constitute the majority of the connected devices in IoT. Moreover, in IoT, the task of authentication is being distributed among nodes at different levels of the network to scalably manage the massive web of various entities. In such scenarios, holding a large CRP database at the verifier end may become impractical.

Intrinsic PUFs [133] are more suitable of IoT since they can be instantiated in off-the-shelf devices with little or no modification to the underlying hardware. All variants of intrinsic PUFs are weak as they offer a limited number of CRPs. The current schemes involving weak PUFs allow a limited number of authentications [134, 135]. To the best of our knowledge, there have been only two weak PUF authentication schemes [136, 137] that allow unlimited mutual authentication sessions. Both schemes employ Fuzzy Extractor to limit the exposure of the PUF response. The scheme in [136] still leaks some information and thus, to ensure security, requires a minimum number of bits (1785 bits to ensure 128-bit security) that may not be available from many intrinsic PUFs. A common limitation of both of these methods is that their proof of correctness must assume certain properties of the distribution of the PUF response, and requires empirical verification. The work in [137] verifies the assumptions by experimental evaluation of the Ring Oscillator (RO) based PUF. However, there is no guarantee that these assumptions will hold for all different PUF designs (or even for RO based PUFs on a different hardware platform). Another drawback of these schemes as well as most other PUF based authentication schemes is that they assume the response to be in binary form. A number of recent intrinsic PUF designs, e.g., the DRAM PUF proposed in [135], generates the PUF response as a set of integers. Authentication of these PUFs requires secure set operations and thus is not compatible with the

schemes presented in [136, 137].

In this paper, we propose an authentication scheme where the prover and the verifier do not reveal a single bit of the PUF response to each other. Thus this scheme supports unlimited mutual authentication sessions even with weak PUFs. Unlike [136] it requires only 237 bits to ensure 128-bit security in an equivalent setting. More importantly, our scheme does not require any assumption on the distribution of the PUF response. In addition, it supports PUF response both in binary form and as a set of integers. Another feature of this protocol is that it allows successful authentication even with the presence of noise in responses at different interrogations of the PUF. Therefore, it does not require any error correction scheme as most of the PUF based protocols [131, 132, 136, 137]. Since one of our primary goals is authentication with intrinsic PUFs we design the scheme such that it can be implemented without any additional hardware.

Our scheme is based on the secure (privacy-preserving) computation of the Hamming distance between the PUF and reference responses from the prover and the verifier, respectively. Interestingly, secure Hamming distance has been employed in the field of biometric authentication. This field faces similar challenges since the biometric keys tend to be noisy and limited in supply, just like the responses of weak PUFs. A number of these works [40, 41] presents protocols for secure computation of Hamming distance in the *malicious* security model, where any party can deviate from the accepted behavior to learn more information about the other party's data or to produce incorrect results. These protocols are proven to be secure in the sense that they do not *leak* any information to any party other than the protocol output. However, we show that the security of the biometric keys can still be breached just from the information held by an individual party.

In our scheme, we first construct an authentication function such that its secure computation does not allow any malicious party to produce a false result or deduce the inputs of the honest party in polynomial (in terms of the number of bits in the response) number of attempts. We then utilize the Yao's Garbled Circuit (GC) [7] to securely compute the authentication function. The original GC protocol was secure in *honest-but-curious* security model, which assumes

that both parties follow the protocol honestly yet may try to learn additional information from the information at hand. However, subsequent enhancements [138, 139, 140, 22] have made it secure in the malicious security model. To add support for the PUFs with integer responses we employ Locality Sensitive Hashing (LSH) [141, 142]. It translates the authentication involving set operations to our Hamming distance based authentication function.

6.3.1 Summary of Contributions

In brief, our contributions of this work are the following.

- We present a mutual authentication scheme for weak intrinsic PUFs based on the secure computation of Hamming distance between the PUF response from the prover and the reference response held at the verifier.
- We prove that our authentication scheme does not allow any malicious party to produce a false result or deduce the inputs of the honest party in polynomial (in terms of the number of bits in the response) number of attempts.
- Our authentication scheme supports PUF response in the classic binary form as well as a set of integers as produced by a number recent intrinsic PUF implementations.
- The authentication scheme supports mutual authentication even with the presence of noise in the PUF response and thus eliminates the need for error correction methods.
- Implementation of the scheme demonstrates the practicality of the design. With maximum allowed fraction of mismatched bits between an authentic pair of responses from the same PUF set to 10%, the protocol execution takes 81 ms on a desktop processor and 487 ms on an embedded processor.

6.3.2 Physical Unclonable Function (PUF)

A Physical Unclonable Function (PUF) is a function whose output (response) depends on both the applied input (challenge) and the unique physical properties of the hardware where it resides. It utilizes the inherent natural physical disorder, e.g., silicon manufacturing variations, of

the device to create a unique signature (fingerprint), of that device. The challenge and response together form one challenge-response pair (CRP). In the exact sense, for a PUF to be used in authentication it must always generate the same response when interrogated multiple times by the same challenge. However, in practice, this criteria is difficult to meet precisely since the response bits are not perfectly reproducible. Transistor noise, as well as various environmental variations (supply voltage, temperature, etc.), introduces noise in the generated response. To ensure usability, the major portion of the response should be stable over multiple interrogations. Some authentication schemes employ error correction methods that can regenerate the reference PUF response given that the number of noisy bits is within a certain limit [143]. In this work, for binary PUF response, we compute the Hamming distance between generated and reference PUF response and compare it against a certain threshold for authentication decision.

There are two broad variants of PUFs: strong PUF and weak PUF [144]. Strong PUF supports a large number (usually exponential in term of a number of challenge bit) of CRPs. This ensures that even if an adversary gains access to a large subset of CRPs, it cannot predict the ones not already known. On the other hand, weak PUFs provide only a limited space of CRPs. In applications involving weak PUFs, it is assumed that its responses are not accessible by adversaries.

In this paper, we are particularly interested in intrinsic PUFs which can be instantiated in off-the-shelf devices without any modification to the hardware. They have limited CRP space and thus are considered weak PUFs. The most widely examined intrinsic PUFs are the one based on Static Random-Access Memory (SRAM) [145, 146, 147]. In SRAM PUFs, the start-up values of the bi-stable SRAM cell constitutes the PUF response which is a binary string. Even though SRAM PUFs have been shown to possess good PUF characteristics, their one major drawback is that since the response is based on start-up values, the authentication can only be performed at boot time or stored in a memory, which subverts the main motivation behind using a PUF.

Recently Dynamic Random Access Memory (DRAM) has been introduced as a PUF construct [148, 149, 135]. The primary advantage of a DRAM PUF is that it can be interrogated

at run-time of the operating system. If the periodic refresh of DRAM cells is turned off or delayed, the charges of the cells decay in a manner unique to each cell resulting in the flipping of the stored value. At a certain delay, the indices of the flipped bits constitute the PUF response. Thus the response of a DRAM PUF is a set of integers rather than a bit stream in generic PUFs. If the DRAM PUF is used for authentication, the similarity of the PUF response with the reference response stored at the verifier is measured using the *Jaccard similarity*. If the response generated by the PUF is R_{PUF} and the reference response saved at the verifier is R_{ref} the Jaccard similarity, $\mathcal{J}$ between these two sets of integers is given by,

$$\mathcal{J} = \frac{|R_{PUF} \cap R_{ref}|}{|R_{PUF} \cup R_{ref}|} \tag{6.6}$$

In Section 6.3.8, we will show how the problem of computation of $\mathcal{J}$ can be translated to the computation of the Hamming distance of two binary strings.

6.3.3 Related Work

Most of the work on PUF based remote authentication involve strong PUFs [127, 128, 129, 130, 131, 132] These works require regular updates of the key and therefore the PUF is required to have a large CRP database. The authors in [150] surveyed the state-of-the-art authentication schemes based on strong PUFs and show that most of them lack formal security analysis and adequate security against various attacks.

There are a few works that deal with remote authentication using weak PUFs. The work in [135] propose a lightweight authentication scheme based on the DRAM PUF construction. In their scheme, the verifier chooses a certain decay time, which acts as the challenge in this case, and sends it to the prover. The prover generates the corresponding response and sends it back to the verifier. The verifier then computes the Jaccard similarity between the received response and the response saved in it CRP database. If the Jaccard index is larger than a certain threshold, the verifier accepts the prover. The drawback of this scheme is that the decay times

have to be monotonically increasing for subsequent authentication sessions. Since there is only limited number of possible decay times, after a certain number of authentication sessions all the PUF responses will be exposed and the PUF will not be useful for further secure authentication sessions.

To the best of our knowledge, there have been two weak PUF based authentication schemes that limit the exposure of the response and thus allow unlimited mutual authentication sessions: the Reverse Fuzzy Extractor presented in [136] and the Trapdoor Computational Fuzzy Extractors presented in [137].

Fuzzy Extractor [143] is used to correct noisy PUF data with the help of a helper data that is generated during the enrollment phase. The main idea in [136] is moving the task of reconstruction of the PUF response from the prover to the verifier. In their reverse fuzzy extractor, the helper data is generated at the prover every time the PUF is interrogated as opposed to generating it only once during enrollment phase in a regular fuzzy constructor. Since the reconstruction of the response requires more computational power than a generation of helper data, this setting improves the overall efficiency of a system with resource-constrained prover and a strong verifier. During authentication, only the helper data is transferred from the prover to the verifier. A legitimate verifier would have the reference PUF response and would be able to regenerate the noisy PUF response generated at that session. Thus authentication is performed without communicating the PUF response. This allows unlimited use of the same response in different authentication sessions. One drawback of this scheme is that it requires a large number of bits to ensure security (1785 bits to ensure 128-bit security) that may not be available from many intrinsic PUFs. Moreover, the helper data inevitably reveals some information about the PUF response [150]. In a regular fuzzy extractor, it is generated only once, while in the reverse fuzzy extractor it is generated in every authentication session resulting in more probability of information leak.

The work in [137] presents a computational fuzzy extractor that can correct $O(m)$ errors in polynomial time. Unlike [136] the confidence information is not exposed in this work. As

a result, it does not require a large number of response bits and can ensure 128-bit security with 128-but PUF response. A limitation common to both [136] and [137] is that their proof of correctness must assume certain properties of the distribution of the PUF response (e.g., it can provide confidence information) that requires empirical verification. The work in [137] demonstrated by experimental results that these assumptions hold for Ring Oscillator (RO) based PUF. However, there is no guarantee that these assumptions will hold for any generic PUF design (or even for a different realization of the RO based PUF). Especially, the intrinsic PUFs has the possibility of having a skewed distribution. Lastly, neither of these two schemes support authentication with PUFs providing integer responses, like the DRAM PUF.

Similar to the PUF responses, biometric keys tend to be noisy. Thus, Hamming distance based authentication has been popular in the field of biometric authentication [41, 40, 151, 152, 153, 154]. While the majority of these work adopt the honest-but-curious security model there have been a number of works that are claimed to be secure in the malicious setting [41, 40]. However, these protocol outputs the Hamming distance in plain text. The security proof of the protocols is based on the premise that a protocol is secure if it does not leak any information to one party other than what can be deduced from her input and the protocol output. However, as we have shown in this paper, revealing the Hamming distance allows an adversary to learn the input of the honest party in a linear number of attempts.

One interesting work in this field is the *bin*HDOT protocol presented in [155]. At the end of this protocol, one party holds a set of variables: $\{Z_l\}_{l=1...L}$ and the other party holds only one of them Z_H where H is the Hamming distance between their inputs. Thus the final result is shared between them. The paper suggests that this can be turned to a secure Hamming distance threshold comparison (similar to the one presented in this paper) by setting $\{Z_l\} = 1$ for $i < T$ and 0 otherwise. However, while the *bin*HDOT protocol is proven to be secure in the malicious setting, this threshold comparison protocol is only secure in the honest-but-curious model. The authors in [155] outline some possible strategies to achieve security in the malicious setting. However, no complete protocol has been developed. This is an example of the fact that while

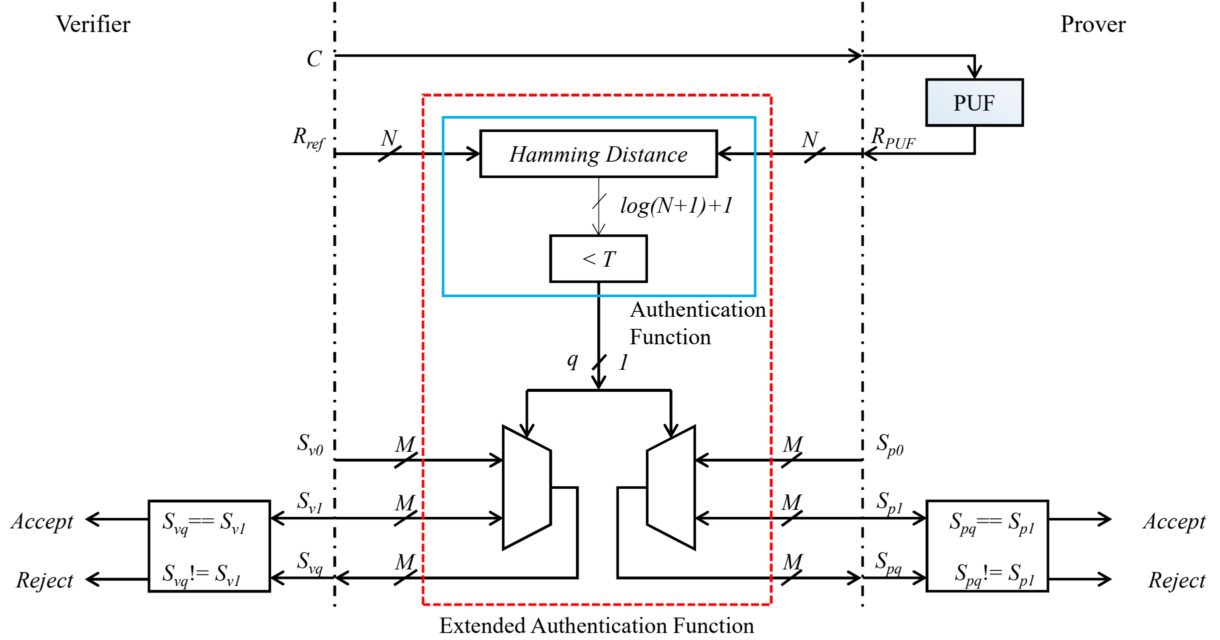


Figure 6.7. Authentication protocol.

designing custom secure protocols may seem efficient, ensuring the security of these protocols is a daunting task. In our design, we first developed a secure authentication function and then executed it with a standard SFE protocol to prevent the possibility of security breach.

6.3.4 Threat Model

Our model consists of three parties, the prover $\mathcal{P}$, the verifier $\mathcal{V}$ and the adversary $\mathcal{A}$. We assume no shared secret between $\mathcal{P}$ and $\mathcal{V}$ other than the CRP (a single CRP will suffice). $\mathcal{P}$ is equipped with a weak PUF. Our scheme allows some noise in the PUF response generated each time by $\mathcal{P}$ and therefore does not require $\mathcal{P}$ to be equipped with any error correction scheme. Further, $\mathcal{P}$ has no secret stored in its non-volatile memory and erases its volatile memory upon exiting the protocol. $\mathcal{A}$ can access the non-volatile memory of $\mathcal{P}$ but not the volatile memory during the time of protocol execution. Consistent with the state of the art schemes, we assume $\mathcal{A}$ to be able to eavesdrop on the communication channel between $\mathcal{P}$ and $\mathcal{V}$. However, unlike previous schemes, we do not assume $\mathcal{V}$ to be trusted. The only distinction we make between $\mathcal{V}$ and $\mathcal{A}$ is the possession of the CRP.

6.3.5 Authentication Function

Figure 6.7 outlines the authentication protocol between $\mathcal{V}$ and $\mathcal{P}$. The steps enclosed within the solid (blue) box constitute the authentication function,

$$q = \mathcal{F}_{auth}(R_{ref}, R_{PUF}, T) = \begin{cases} 1, & HD(R_{ref}, R_{PUF}) < T \\ 0, & otherwise \end{cases} \tag{6.7}$$

where, $HD(.,.)$ denotes the Hamming Distance. The input to the authentication function from $\mathcal{V}$ is the reference PUF response R_{ref}, an N-bit binary string stored in its database. The input from $\mathcal{P}$ is the PUF response R_{PUF}, an N-bit binary string extracted from the PUF. The first step of the function is computing the Hamming distance between R_{ref} and R_{PUF}. If PUF responses were free from noise or in presence of error correcting code, the Hamming distance would be zero for a genuine PUF response. However, to allow a generic PUF without any error correction, we compare the resultant Hamming distance with a threshold value, T. Here, T is a publicly known parameter agreed upon by $\mathcal{V}$ and $\mathcal{P}$ before the start of the protocol (during the initialization phase, Section 6.3.6). The choice of T will depend on the characteristics of the PUF and the acceptable error margin. The 1-bit output q of the comparator indicates whether or not the PUF response is close enough to the reference response saved at the verifier.

We employ the Authenticated Garbling protocol [22] to ensure the privacy of the inputs of $\mathcal{V}$ and $\mathcal{P}$. We now take a closer look into the security of this protocol. As discussed in Section 2.4, the final output is XOR-shared between the two participating parties, and they have to authenticate their respective shares to prevent alteration in the middle of the protocol execution. The parameters of the authentication function $\mathcal{F}_{auth}$ are set such that the probability of success of an adversary $\mathcal{A}$ with an incorrect PUF response is infinitesimally small (2^{-128}, see Section 6.3.9 for details). Therefore, if the authentication function $\mathcal{F}_{auth}$ is evaluated through the Authenticated Garbling protocol, the adversary would already know the value of the output q (0, in this case) with success rate close to unity. Therefore, $\mathcal{A}$ could flip her share of the bit q, which would result

in flipping the final value of q from 0 to 1 and making the honest party accept her as a legitimate $\mathcal{V}$ or $\mathcal{P}$. The Authenticated Garbling protocol is able to prevent alteration of the shares in the middle of the protocol execution. In this specific case, the final output is independent of the input of the honest party and is known to the adversary even before the execution starts.

To prevent the above scenario, instead of directly using q as the output, it is used as the selector bit of two multiplexers to select from two pairs of random M bit strings. This constitutes the *extended* authentication function,

$$\{S_{vq},\ S_{pq}\} = \mathcal{F}_{ext_auth}(R_{ref}, S_{v0}, S_{v1}, R_{PUF}, S_{p0}, S_{p1}, T) \tag{6.8}$$

The steps involved in this function is enclosed within the dotted (red) box in Figure 6.7. This extended authentication function is computed through the Authenticated Garbling protocol. Along with R_{ref} and R_{PUF}, the inputs to the extended authentication function from $\mathcal{V}$ is a pair of M-bit random nonces, S_{v0} and S_{v1}. Similarly, the inputs from $\mathcal{P}$ is a pair of M-bit random nonces, S_{p0} and S_{p1}. The 1-bit output q of the comparator is used as the selector bit of the two multiplexers with inputs $\{S_{v0}, S_{v1}\}$ and $\{S_{p0}, S_{p1}\}$. The outputs of the multiplexers S_{vq} and S_{pq} are revealed to $\mathcal{V}$ and $\mathcal{P}$, respectively. The final decision is made through the local comparison. Each party accepts the other party if and only if the received outputs entirely matches the S_{v1} (Verifier side) or S_{p1} (Prover side). Unlike the authentication function in Equation 6.7, the outputs of this extended authentication function depend on the inputs from both parties even with an incorrect PUF response. $\mathcal{A}$ will now have to correctly guess S_{v1} or S_{p1} which has a success rate of 2^{-M}.

6.3.6 Protocol Initialization

Similar to all PUF based authentication protocols, we assume that the initialization is performed in a secure environment. $\mathcal{V}$ sends a set of challenges (at least one) to $\mathcal{P}$ and stores the responses sent back by $\mathcal{P}$ in her database. Note that the stored responses are considered secret

owned by $\mathcal{V}$, while the challenges are public. $\mathcal{P}$ stores no secret data in its memory. $\mathcal{P}$ and $\mathcal{V}$ agree on the threshold value T which is publicly known and is stored on the non-volatile memory of both parties. In addition to these, the netlist of the authentication function required by the GC protocol is also stored on the non-volatile memory. The netlist can be generated by $\mathcal{V}$, $\mathcal{P}$, or an independent issuer. As explained in Section 2.4, the netlist is publicly known, independent of either $\mathcal{V}$ or $\mathcal{P}$ and only has to be generated once during the initialization phase.

6.3.7 Protocol for Binary Response

We now describe our protocol for PUFs that generate a binary string response. The protocol is denoted as π_{auth}. In the next section, we describe how this protocol can be extended to be compatible with the PUF response as a set of integer indices. In π_{auth}, the extended authentication function $\mathcal{F}_{ext_auth}$ is computed through Authenticated Garbling [22]. This ensures that the inputs and outputs of this function remain private to the respective parties and is not revealed to the other party (or an eavesdropper). It also ensures the correctness of the output even if one of the parties are corrupted.

Authentication protocol π_{auth} between the verifier $\mathcal{V}$ and the prover $\mathcal{P}$:

Input: $\mathcal{V}$ inputs the N-bit reference response R_{ref}, and two M-bit random nonces S_{v0}, S_{v1}. $\mathcal{V}$ inputs the N-bit PUF response R_{PUF}, and two M-bit random nonces S_{p0}, S_{p1}. The challenge C and the threshold T is public and known by both parties.

Output: Both parties receive a one-bit output ACCEPT/REJECT indicating whether or not the other party is authenticated.

The protocol:

i $\mathcal{V}$ sends the challenge C to $\mathcal{P}$. If there is only one CRP then C can be stored in a non-volatile memory of $\mathcal{P}$ and this step can be omitted.

ii $\mathcal{P}$ applies the challenge C to the PUF and generates the response R_{PUF}. In addition, $\mathcal{P}$ generates two random M-bit strings S_{p0} and S_{p1}.

iii $\mathcal{V}$ generates two random M-bit strings S_{v0} and S_{v1}.

iv $\mathcal{V}$ and $\mathcal{P}$ perform the Authenticated Garbling protocol on the extended authentication function $\mathcal{F}_{ext_auth}$ (Figure 6.7) with $\mathcal{V}$ as the garbler and $\mathcal{P}$ as the evaluator. The inputs to the authentication function from $\mathcal{V}$ are the N-bit reference PUF response R_{ref} and the two M-bit random nonces S_{v0} and S_{v1}. Similarly, the inputs from $\mathcal{P}$ are the N-bit PUF response R_{PUF} and two M-bit random nonces S_{p0} and S_{p1}.

v $\mathcal{P}$ sends his share of S_{vq} to $\mathcal{V}$ so that $\mathcal{V}$ can XOR it with her share and learn the actual value.

vi Similarly, $\mathcal{V}$ sends her share of S_{pq} to $\mathcal{P}$ so that $\mathcal{P}$ can XOR it with his share and learn the actual value.

vii $\mathcal{V}$ locally compares S_{vq} with S_{v0} and S_{v1}. If $S_{vq} = S_{v1}$, $\mathcal{V}$ accepts $\mathcal{P}$. Otherwise, $\mathcal{V}$ rejects $\mathcal{P}$ and aborts.

viii $\mathcal{P}$ locally compares S_{pq} with S_{p0} and S_{p1}. If $S_{pq} = S_{p1}$, $\mathcal{P}$ accepts $\mathcal{V}$. Otherwise, $\mathcal{P}$ rejects $\mathcal{V}$ and aborts.

6.3.8 Extension for Integer Response

As we discussed in Section 2.4, for the DRAM PUF we need to translate the authentication based on Jaccard similarity to the authentication based on Hamming Distance. For this purpose, we utilize Locality Sensitive Hashing (LSH) [141, 142].

LSH is a family of functions that map the input domain to the output domain (hash) with the following condition: *the probability that hashes of two inputs are equal (collision) is higher for similar inputs than non-similar ones*. The similarity between inputs can be quantified using different similarity metrics such as Jaccard or Cosine. More formally, if the collision probability $Pr_{\mathcal{H}}(h(x) = h(y))$ for a hash family $\mathcal{H}$ is a monotonically increasing function of the similarity,

$Sim(x, y)$, the hash family $\mathcal{H}$ is a valid LSH family

$$Pr_{\mathcal{H}}(h(x) = h(y)) = f(Sim(x, y)), \tag{6.9}$$

where $f(.)$ is a monotonically increasing function. To be consistent with [135] which presents the most efficient DRAM PUF to date, we use the Jaccard similarity index $\mathcal{J}$ [156] in the authentication method presented in this paper. The Jaccard similarity between two given sets x, $y \subseteq \Omega = \{1, 2, ..., |\Omega|\}$ is defined as

$$\mathcal{J} = \frac{|x \cap y|}{|x \cup y|} \tag{6.10}$$

Minwise hashing (MinHash) [157] is the LSH that preserves the Jaccard similarity and is defined as follows

$$h_{\pi}^{min}(x) = \min(\pi(x)), \tag{6.11}$$

where $\pi : \Omega \longrightarrow \Omega$ is a random permutation applied to the given set x. The hash is the minimum value of the permuted set. For example, given the set $x = \{1, 2, 5\} \subset \Omega = \{1, 2, 3, 4, 5\}$ and the random permutation

$$\pi : 1 \longrightarrow 5, 2 \longrightarrow 3, 3 \longrightarrow 1, 4 \longrightarrow 2, 5 \longrightarrow 4,$$

set x is mapped to $\pi(x) = \{5, 3, 4\} = \{3, 4, 5\}$, hence, $h_{\pi}^{min}(x) = 3$. For any two sets x and y, by an elementary probability argument (see [158]) it can be shown that

$$Pr\{h_{\pi}^{min}(x) = h_{\pi}^{min}(y)\} = \frac{|x \cap y|}{|x \cup y|} = \mathcal{J}. \tag{6.12}$$

MinHash is a valid LSH since the function $f(.)$ in Equation 6.9 is equal to $f(\alpha) = \alpha$ which is a monotonically increasing function. Please note that MinHash maps an input set to an integer value $h_{\pi}^{min}(x) : S \subseteq \Omega \mapsto i \in \{1, 2, ..., |\Omega|\}$. However, from the computation and storage consumption perspective, it is preferable to map the output integer to only a 1-bit output. This can be realized by the universal hash functions $h_{univ} : \mathbb{N} \mapsto \{0, 1\}$. One of the popular approaches is to take

only the least significant bit of MinHash [159]. Therefore, 1-bit MinHash can be realized as $h_\pi^{min,1bit}(x) = h_{univ}(h_\pi^{min}(x)) = h_\pi^{min}(x) \bmod 2$. In order to identify whether $h_\pi^{min,1bit}(x)$ is a valid LSH or not, we need to compute the collision probability. The probability that two sets x and y have equal MinHash values is $\mathcal{J}$ (Equation 6.12) in which case $h_\pi^{min,1bit}(x) = h_\pi^{min,1bit}(y)$. If $h_\pi^{min}(x) \neq h_\pi^{min}(y)$ (with probability $1-\mathcal{J}$), there is a 50% chance that 1-bit hashes would collide (symmetry between outcome events). As a result

$$Pr\{h_\pi^{min,1bit}(x) = h_\pi^{min,1bit}(y)\} = \mathcal{J} \times 1 + (1-\mathcal{J}) \times \frac{1}{2} = \frac{\mathcal{J}+1}{2}. \tag{6.13}$$

Consequently, 1-bit MinHash is also a valid LSH. One can repeat the computation ℓ times with ℓ different random permutations to create an ℓ-bit LSH embedding. We denote the ℓ-bit LSH embedding of a set x as $LSH_{min}^{\ell}(x)$. Given two sets x and y with similarity of $Sim(x,y) = \mathcal{J}_0$, the number of bit-matches between $LSH_{min}^{\ell}(x)$ and $LSH_{min}^{\ell}(y)$ is $\frac{\mathcal{J}_0+1}{2} \times \ell$ on average. The uncertainty comes from the fact that LSH is a probabilistic embedding. More precisely, $E[NumBitMatch(LSH_{min}^{\ell}(x), LSH_{min}^{\ell}(y))] = \frac{\mathcal{J}_0+1}{2} \times \ell$, where $E[.]$ is the expected value of a random variable. One can express this formula using Hamming Distance (HD)

$$E_{HD}^{\ell}(x,y) = E[HD(LSH_{min}^{\ell}(x), LSH_{min}^{\ell}(y))] = \frac{1-\mathcal{J}_0}{2} \times \ell. \tag{6.14}$$

For instance, if $\mathcal{J}_0 = 0.9$, the Hamming distance between the 64-bit LSH embeddings of x and y ($E_{HD}^{64}(x,y)$) is 3.2. Therefore, if the Jaccard similarity of 0.9 or higher is accepted as the verified response, one can similarly verify the hash of a response if it passes the HD threshold of 3.

6.3.9 Security of the Authentication Function

The security of the Hamming distance based schemes relies on the fact that the Hamming distance sums up the difference between the bits of the two input binary strings and thus holds no information about individual bits. While this is true for a single execution, in the case of

multiple executions, it is possible to deduce information about individual bits of one input by adaptively varying the other input and observing the changes in the resulting Hamming distance. Any authentication scheme based on the Hamming distance should ensure that the authentication keys cannot be learned in a polynomial (in terms of the number of bits in the keys) number of executions.

We start with a more generic version of the authentication function,

$$Q = \mathcal{F}^{\mu}_{gen_auth}(R_{ref}, R_{PUF}, T) \tag{6.15}$$

where, Q is the left most μ bits ($1 \leq \mu \leq log(N)+1$) [1] of $HD(R_{ref}, R_{PUF}) - T$. The authentication function $\mathcal{F}_{auth}$ provided in Equation 6.7 is a special case of $\mathcal{F}_{gen_auth}$ with $\mu = 1$, since the output of the comparator is essentially the carry bit of the subtraction result.

Hamming distance has been employed in biometric authentication protocols [40, 41] which deals with noisy signatures similar to the PUF responses. However, these protocols generally output the Hamming distance itself and the comparison with the threshold is performed locally. The authentication function employed in these protocols is another special case of the one provided in Equation 6.15 with $\mu = log(N)+1$ and $T = 0$. The security analysis of these protocols prove that a malicious adversary cannot obtain more information than what can be generated from the information she has (i.e., her input and received function output). However, even if the protocol is assumed to be secure by definition, we show in this section that it is possible for the adversary to deduce the input of the honest party in $O(N)$ attempts. This is why we start with the generic authentication function given in Equation 6.15 and show that it is secure in the special case of

$$\mathcal{F}_{auth}(R_{ref}, R_{PUF}, T) \equiv \mathcal{F}^{\mu=1}_{gen_auth}(R_{ref}, R_{PUF}, T) \tag{6.16}$$

[1] For fractional values of $log(.)$, we take the nearest integer smaller than $log(.)$, i.e., $\lfloor log(.) \rfloor$. For simplicity, we denote $\lfloor log(.) \rfloor$ as $log(.)$ throughout the paper.

Let us define another variable,

$$\nu = log(N) + 1 - \mu \tag{6.17}$$

Thus, ν is the number of bits hidden from the subtraction result of $HD(R_{ref}, R_{PUF}) - T$.

In the following analysis, we examine the effect of ν on the security of the authentication function. We assume that the adversary $\mathcal{A}$ with input R'_{PUF} is posing as the prover $\mathcal{P}$ and interacts with an ideal simulator Σ that computes the function $\mathcal{F}^{\mu}_{gen_auth}(R_{ref}, R'_{PUF}, T)$. Σ holds R_{ref} and is secure by definition, i.e., it does not leak any information regarding R_{ref}, except the output of the function. $\mathcal{A}$ tries to deduce R_{ref} by adaptively updating her input R'_{PUF}.

- Initially, we set $\nu = 0$ and $T = 0$. To deduce R_{ref} in the minimum number of attempts, $\mathcal{A}$ should update her input with the finest resolution. Therefore, initially, she flips a single bit of her input R'_{PUF} at index n in 2 consecutive attempts. If the values of the output Q provided by Σ increases, then the n-th bit of R'_{PUF} at the first attempt was equal to the n-th bit of R_{ref}, since flipping that bit resulted in an increased Hamming distance. In this setting, $\mathcal{A}$ will require $2N$ attempts to deduce all N bits of R_{ref} (Thus, the protocols of [40, 41] are not secure since they can be broken in $O(N)$ attempts).
- If ν is set to 1, i.e., the least significant bit of the Hamming distance is hidden from $\mathcal{A}$, flipping only 1 bit will not produce any difference in the values of Q. Therefore, $\mathcal{A}$ is forced to flip 2 bits of R'_{PUF}. Since there are 4 possible combinations, she will need 4 attempts to learn 2 bits of R_{ref}, and a total of $4 \times N/2$ attempts to learn all N bits.
- We now consider an arbitrary value of ν. The Hamming distance between two W-bit binary strings is $log(W) + 1$ bits. However, if we keep the first string fixed, for only one of the 2^W possible instances of the second string the most significant bit (bit index $log(W) + 1$) of the Hamming distance will be set to 1 when the second string is bit-by-bit inverse of the first one. Similarly, to observe a change in the $(\nu + 1)$-st bit of the Hamming distance, $\mathcal{A}$ will require to alter groups of 2^{ν} bits together, and thus will need $2^{2^{\nu}}$ attempts. In total, she

will need $2^{2^{\nu}} \times N/2^{\nu}$ attempts to learn all the bits of R_{ref}.

- Finally, for $\nu = log(N)$, thus $\mu = 1$ as employed in the authentication function of Equation 6.7, $\mathcal{A}$ will need $2^{2^{log(N)}} \times N/2^{log(N)} = 2^N$ attempts to learn all the bits of R_{ref}. Thus the number of attempts is exponential in N.

Effect of the Threshold T. So far we have assumed the threshold $T = 0$, which is ideal but not practical due to the noise present in the PUF response. For $T > 0$, it will suffice for $\mathcal{A}$ to learn any $N - T$ out of N bits correctly, irrespective of the bit indices that are correct. Since the distribution of the bits of the PUF responses is unknown to $\mathcal{A}$, the process of forming these N bit binary strings follows a binomial distribution [160] which is used to model the number of successes when sampling with replacement. Let w be the number of bits guessed correctly. Assuming each bit of R_{ref} has equal probability ($= 1/2$) of being 0 or 1,

$$Pr(w \geq N - T) = \frac{1}{2^N} \sum_{n=N-T}^{N} \binom{N}{n} \tag{6.18}$$

Note that the nominator in Equation 6.18 is a polynomial in N and the denominator is an exponential in N. Therefore, the number of attempts required by $\mathcal{A}$ to authenticate with a high probability is still exponential in N. To ensure 128 bit security, we need

$$Pr(w \geq N - T) \leq 2^{-128} \tag{6.19}$$

We set T as a fraction t of N, i.e., $T = \lceil tN \rceil$; $0 < t < 1$. Solving inequality 6.19 for $t = 0.05, 0.1, 0.15$ yields $N = 181, 237, 320$, respectively. The threshold fraction t, which denotes the maximum fraction of mismatched bits between an authentic pair of R_{PUF} and R_{ref}, is the only parameter that controls the total execution time of the protocol.

Minimum Set Size for PUFs with Integer Responses. For the case of PUFs with integer responses, along with the minimum bit-length for LSH encoding, we need to set a minimum size of the sets so that the probability of an adversary successfully guessing the response set is

sufficiently low. Let S be the set guessed by $\mathcal{A}$, u be the number of elements guessed correctly, and v be the number of elements guessed incorrectly. The Jaccard similarity, $\mathcal{J}$ can be computed as

$$\mathcal{J} = \frac{|S| - v}{|S| + v}, \tag{6.20}$$

If the maximum number of incorrect guess is v_M, then the minimum value of Jaccard similarity, $\mathcal{J}_{min}$ is given by

$$\mathcal{J}_{min} = \frac{|S| - v_M}{|S| + v_M}, \tag{6.21}$$

For a given $\mathcal{J}_{min}$, we have

$$v_M = \frac{1 - \mathcal{J}_{min}}{1 + \mathcal{J}_{min}} \times |S|, \tag{6.22}$$

The process of choosing integers to form the sets of responses follows the multivariate hypergeometric distribution [161] that models sampling without replacement. Since each element (the integer indices) is represented only once in the urn,

$$\begin{aligned} Pr(\mathcal{J} \geq \mathcal{J}_{min}) = Pr(v \leq v_M) &= Pr(u \geq |S| - v_M) \\ &= \sum_{n=|S|-v_M}^{|S|} \frac{\binom{|\Omega|-|S|}{|S|-n} \times \binom{|S|}{n}}{\binom{|\Omega|}{|S|}} \end{aligned} \tag{6.23}$$

To ensure 128 bit security we need

$$Pr(\mathcal{J} \geq \mathcal{J}_{min}) \leq 2^{-128} \tag{6.24}$$

In the DRAM PUF construction of [135] each logical PUF has the size of 32KB, which gives $|\Omega| = 32 \times 8 \times 2^{10}$. Solving inequality 6.24 for $\mathcal{J}_{min} = 0.9$ yields $|S| \geq 10$.

6.3.10 Security of the Authentication Protocol

We now analyze the security of the authentication protocol, π_{auth}. We again assume that the adversary $\mathcal{A}$ with input R'_{PUF} is posing as the prover $\mathcal{P}$. $\mathcal{A}$ interacts with an ideal simulator

Σ. Unlike the previous section, in this case, Σ does not communicate with the verifier $\mathcal{V}$, and therefore have no knowledge about the reference response R_{ref}. We prove the security of π_{auth} by showing that Σ is able to generate a view that is indistinguishable from the view of the adversary in a real execution of the protocol, π_{auth} with $\mathcal{V}$. This would imply that $\mathcal{A}$ learns no information about the input of $\mathcal{V}$ from the real protocol [162]. In this analysis, we utilize the hybrid model presented in [163]. According to this model, if a protocol is proven to be secure in the right setting, it suffices to assume that the parties have access to a trusted party that computes that functionality. Since the Authenticated Garbling protocol of [22] is proven to be secure in the malicious model, we assume that the parties have access to a trusted party that computes this functionality.

The ideal simulator Σ works as follow:

i Σ sends the challenge, C to $\mathcal{A}$. Since C is considered public, Σ does not need communication with $\mathcal{V}$ to learn it.

ii Σ receives the the response R'_{PUF}, two random M-bit strings S_{p0} and S_{p1} from $\mathcal{A}$

iii Σ generates a N-bit random string R'_{ref} and two random M-bit strings S_{v0} and S_{v1}.

iv Σ performs the Authenticated Garbling protocol on the extended authentication function $\mathcal{F}_{ext_auth}$ through a trusted party. The inputs to the extended authentication function from Σ is the N-bit random string R'_{ref} and two the M-bit random nonces, S_{v0} and S_{v1}.

v Σ receives $\mathcal{A}$'s share of S_{vq}.

vi Σ sends its share of S_{pq} to $\mathcal{A}$

Since none of R'_{ref} from Σ and R'_{PUF} from $\mathcal{A}$ are authentic, $\mathcal{A}$ observes with a high probability that the authentication has failed, according to the analysis in Section 6.3.9. Since her own input R'_{PUF} is not authentic, this is the expected result from her side. Therefore, she is not able to distinguish between a real execution of the protocol π_{auth} with $\mathcal{V}$ and an ideal execution by the

simulator Σ, which does not communicate with $\mathcal{V}$. This proves that the π_{auth} does not leak any information about the inputs of $\mathcal{V}$.

Note that the security proof could be stronger if we assumed $\mathcal{A}$ "corrupts" $\mathcal{P}$ as opposed to "posing as" $\mathcal{P}$, in which case she would posses the actual PUF response R_{PUF}. However, R_{PUF} is just a noisy version of R_{ref}. Therefore, $\mathcal{A}$ would learn nothing new from the protocol in that case.

6.3.11 Generating GC Netlist

To execute securely through the Authenticated Garbling [22] protocol (or any garbled circuit based protocol), the extended authentication function shown in Figure 6.7 needs to be represented as a netlist of Boolean logic gates. Due to free-XOR [26] described in Section 2.4, optimizing the netlist for GC requires minimizing the number of non-XOR gates. We design the function in Verilog HDL and compile by Yosys Open Synthesis Suite [164] with the TinyGarble [6] circuit synthesis library for GC to generate the netlist. Even though TinyGarble is the most efficient tool to generate the GC netlist, the GC execution protocol supported by this framework is only secure in the semi-honest setting. Therefore we garble/evaluate the generated netlist through the realization of the Authenticated Garbling protocol provided in the EMP-toolkit [35]. This protocol is secure in the malicious setting as demanded by our authentication scheme.

Table 6.4 shows the number of non-XOR gates required by the different components of the extended authentication function for different values of threshold fraction t. The number of bits in the PUF response N and the threshold T are set according to the computations in Section 6.3.9. M is set to 128, the value of the security parameter k employed in recent works [21, 22] in Secure Function Evaluation (SFE). Note that the total number of non-XOR gates in the netlist generated by Yosys is smaller than the sum of the number of non-XOR gates in all components. This is because the synthesis tools preform optimizations on the entire circuit to minimize the cost function, which in this case is the number of non-XOR gates.

We have created a parser to automatically convert the netlist from TinyGarble in the

format supported by the EMP-toolkit. The size of the netlist files for different settings is shown in Table 6.4. The netlist needs to be generated only once and stored on the non-volatile memory of both parties.

Table 6.4. The numbers of non-XOR gates in the generated netlist for different values of threshold fraction t

t	N	T	M	Number of non-XOR gates					Size(KB)
				Hamming	Comparator	MUX	Total	Total†	
				$N-1$	$log(N)+1$	$2M$	$N+log(N)+2M$	Gen. by Yosys	
0.05	181	10	128	180	8	256	444	439	42
0.10	237	24	128	236	8	256	500	494	51
0.15	320	48	128	319	9	256	584	582	64

† The synthesis tool perform optimization on the entire circuit to reduce the number of gates.

6.3.12 Implementing LSH

As we discussed in Section 6.3.8, in order to create a 1-bit LSH from a set, we need to perform a random permutation on the input set. Computing an l-bit LSH, in fact, requires ℓ random permutations which accounts for the costliest part of computing the LSH embedding. However, recent advances in Minwise hashing [165, 166, 167] make it possible to extract more than one bit from a single permuted set. The general idea behind these methods is that one can partition the universal set (Ω) into N_p different pieces and perform Minwise hashing for each partition separately. In other words, once we have permuted the input set, we output the minimum value in each section as the MinHash value and mod 2 as its 1-bit LSH. Figure 6.8 illustrates this idea for a sample set of $S = \{0,3,7,\ldots,|\Omega|-2\}$. This technique enables us to extract N_p 1-bit LSH from single permutation which in turn, reduces the computation and memory usage by almost a factor of N_p. As a result, the overall number of random permutations required to generate a ℓ-bit LSH is $\lceil \frac{\ell}{N_p} \rceil$.

We present two different approaches to compute the permuted version of a set. The first approach is to pre-compute all $\lceil \frac{\ell}{N_p} \rceil$ permutations and store them in memory. In order to permute a set S, one needs to perform only $|S|$ memory accesses. This approach is very fast ($O(|S|)$ read

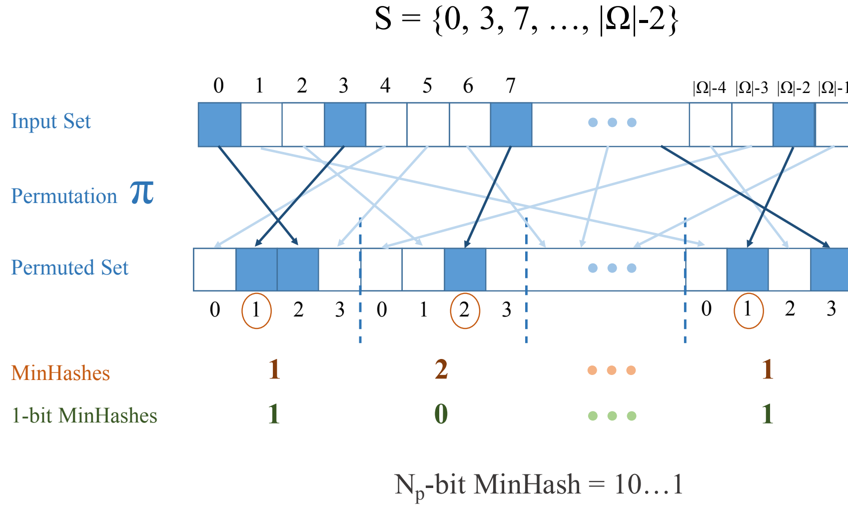


Figure 6.8. Extracting multiple LSH bits from single random permutation π.

operations) but it needs to compute and store all random permutations which take $O(|\Omega|)$ of memory. The second approach only requires $O(|S|)$ of memory but the computation is slower which we describe next.

Utilizing HW-Based Pseudo-Random Permutation (PRP). . We propose to employ PRP such as Advanced Encryption Standard (AES) to efficiently perform the permutation. Since almost all modern processors have AES-NI in their instruction set (ISA), the fast HW-based permutation can be implemented using AES. More precisely, one of the parties randomly generates a 128-bit key (K) and announce it publicly. Each time a party wants to permute a set, he computes $\text{AES}_K(S(i))$ for $i = 1, 2, ..., |S|$ where $S(i)$ denotes the ith member of S. This approach does not support scenarios where $|\Omega| > 2^{128}$ but this limit is far beyond our requirements. Please note that AES only operates on 128-bit input blocks, whereas, the members of set S are represented with $\log_2(|\Omega|)$ number of bits. However, this is not an issue since instead of partitioning the universal set Ω, we partition the output range of AES. This results in same hash results since AES is a uniform random permutation (each input has equal chance to take any output value).

Overall Cost and Complexity. We summarize the complexity of computational time

and memory utilization for both aforementioned approaches in Table 6.5.

Table 6.5. Computational time and memory utilization complexity for two different implementations of LSH.

	Time	Memory
Pre-computing Permutations	$\mathcal{O}(\lvert S\rvert \frac{\ell}{N_p})$ Read Ops	$\mathcal{O}(\lvert\Omega\rvert \frac{\ell}{N_p})$
HW-based PRP	$\mathcal{O}(\lvert S\rvert \frac{\ell}{N_p})$ AES Ops	$\mathcal{O}(\lvert S\rvert)$

6.3.13 Evaluation Settings

We employ two platforms to evaluate the authentication protocol. Platform 1 is an Intel Core i7-2600 CPU @3.4GHz with 12 GB of memory running Ubuntu 14.04. Platform 2 is an Intel Atom E3815 @1.46 GHz with 2GB of memory running Lubuntu 16.04.1. The protocol is evaluated in the following two settings:

- Setting A: To assess the best case capability, we run the protocols for both $\mathcal{V}$ and $\mathcal{P}$ on Platform 1.
- Setting B: To emulate the practical scenario of a verifier with high computational power and a resource-constrained prover, we run the protocol for $\mathcal{V}$ on Platform 1 and the protocol for $\mathcal{P}$ on Platform 2.

6.3.14 Evaluation of the Authentication Protocol

The GC protocol is run through the realization of the Authenticated Garbling [22] in the EMP-toolkit [22]. We first evaluate the timing of the protocol involving only the extended authentication function, without LSH. The Authenticated Garbling protocol consists of three phases: (1) *Set up*: generate correlated randomness between $\mathcal{V}$ and $\mathcal{P}$ that are used during the last (online) phase for information-theoretic authentication of different values. (2) *Function-independent pre-processing*: independent of the inputs from $\mathcal{V}$ or $\mathcal{P}$ or the extended authentication function. (3) *Function-dependent pre-processing*: dependent of the authentication function, but not the inputs. (4) *Online phase*: dependent on both the extended authentication function and the

inputs from $\mathcal{V}$ and $\mathcal{P}$. Tables 6.6a and 6.6b show the the number of clock cycles and time on both $\mathcal{V}$ and $\mathcal{P}$ at each stage of the protocol in the two settings for different values of the threshold fraction t. Each timing measurement is averaged over 10 instances. The total time on the prover side for $t = 0.1$ is 399 ms in Setting B, which emulates the real life scenarios. To put this time into context, the interrogation time of the DRAM PUF [135] is in the range of minutes. Therefore, the protocol execution time is negligible compared to the response generation time of the PUF.

Table 6.6. Timing evaluation of the authentication protocol in the two settings for different values of the threshold fraction t.

(a) Setting A: Both $\mathcal{V}$ and $\mathcal{P}$ on powerful platform.

Stage	t = 0.05				t = 0.10				t = 0.15			
	Verifier		Prover		Verifier		Prover		Verifier		Prover	
	cc	ms	cc	ms	cc	ms	cc	ms	cc	ms	cc	ms
Set up	1.7E8	77.25	1.6E8	74.37	1.6E8	76.01	1.6E8	73.01	1.6E8	76.37	1.5E8	72.24
Func. Indep.	9.9E6	4.52	1.7E7	7.71	9.9E6	4.53	1.7E7	7.81	1.1E7	5.08	2.0E7	9.48
Func. Dep.	2.4E6	1.10	1.7E6	0.79	2.1E6	0.96	1.5E6	0.71	2.4E6	1.11	2.0E6	0.92
Online	2.4E5	0.11	2.9E6	1.33	3.1E5	0.14	2.8E6	1.30	3.6E5	0.17	3.4E6	1.57
Total	1.8E8	82.99	1.8E8	84.19	1.8E8	81.66	1.8E8	82.83	1.8E8	82.72	1.8E8	84.20

(b) Setting B: $\mathcal{V}$ on powerful platform and $\mathcal{P}$ on resource-constrained platform.

Stage	t = 0.05				t = 0.10				t = 0.15			
	Verifier		Prover		Verifier		Prover		Verifier		Prover	
	cc	μs	cc	μs	cc	μs	cc	μs	cc	μs	cc	μs
Set up	6.8E8	309.88	4.3E8	295.99	6.8E8	311.18	4.3E8	296.17	6.8E8	313.25	4.3E8	296.30
Func. Indep.	1.3E8	61.78	1.0E8	69.33	1.9E8	90.66	1.4E8	98.19	2.0E8	93.24	1.4E8	100.99
Func. Dep.	1.8E7	8.35	4.5E6	3.06	1.9E7	8.70	4.8E6	3.33	2.0E7	9.40	5.7E6	3.89
Online	5.3E6	2.45	3.7E6	2.58	5.4E6	2.47	3.2E6	2.23	5.5E6	2.53	3.8E6	2.63
Total	8.4E8	382.46	5.4E8	370.97	9.0E8	413.01	5.8E8	399.93	9.2E8	418.42	5.9E8	403.81

6.3.15 Evaluation of Protocol for Integer Response

To evaluate the timing of our protocol while authenticating the DRAM PUF with integer responses we need to add the time that LSH computation takes. The time and memory utilization of two different approaches for hash computation is illustrated in Table 6.5. For the actual timing results, we use the HW-based PRP method. As we discussed, we partition the output space of AES into N_p different pieces. As can be seen from Table 6.5, by increasing N_p, one can reduce the overall computation time. However, there is an upper bound limit on N_p [165] that depends

on $|\Omega|$ and $|S|$. In our setting, the limit is 300 hashes. Therefore, having $\ell < 300$ means that we can create ℓ-bit LSH embedding using a single permutation ($\lceil \frac{\ell}{N_p} \rceil = 1$). The overall hash computation time is therefore bounded by the number of AES invocations which is $|S|$.

The DRAM PUF presented in [135] employed logical PUF constructions on 32KB segments of the DRAM. To simulate this construct, we take Ω as the set of integers from 0 to $(32 \times 8 \times 2^{10} - 1)$. We consider the PUF response and the reference response stored at the verifier to be sets of 300 integers which are the subset of Ω. The total number of clock cycles for LSH are 3.13E+05 on Platform 1 and 1.04E+06 on Platform 2. These translate to 92μs and 867μs respectively.

6.4 Privacy Preserving k-Nearest Neighbor Search

Recently ride sharing apps like Uber, Lyft have become popular both as cheap rides and a source of income by providing rides. With the emerge of these services concern over the privacy of both the riders and the drivers have gained attention. To ensure location privacy of both the parties, we need to design a system that allows the rider to search for the nearest car without revealing her location to the service provider while ensuring that the rider only knows about the few nearby drivers.

Prior to the publication of this work [20], the only available implementation of the privacy-preserving similarity search using the GC protocol was for the 1-NN search, where the circuit size was linearly increasing with the dataset size [168]. This increase is due to the fact that conventional combinational logic representation that was employed in that implementation is not scalable.

We present the first efficient, practicable, and scalable methodology for privacy-preserving k-NNS based on the Yao's GC protocol. It utilizes the sequential circuit description for GC, which was first introduced by our GC framework TinyGarble [6], instead of the conventional combinational representation. It also benefits from the custom libraries presented in TinyGar-

ble [6]. As a result, we can store the GC and perform the privacy preserving k-NNS with an unprecedented efficiency.

In our implementation, the rider, Alice has a query q, which is her location and the service provider, Bob has a dataset S containing the location of the available drivers in that area. They want to jointly compute the k nearest neighbors of q in S such that Bob does not learn anything about q and Alice does not learn anything about S except the nearest drivers.

Our work reduces the size of the required memory for GC from $O(nw)$ to $O(w)$ compared with the best known GC implementation of 1-NN [168]. Our scalable implementation requires a memory in the order of $O(kw)$ for k-NNS search. Note that k-NNS was impracticable earlier (for $k > 1$) due to the linear growth of the combinational representation. Proof-of-concept implementation of privacy preserving k-NNS on an Intel processor with $w = 31, k = 8$ requires only 80KB of memory.

6.4.1 Summary of Contributions

In brief, our contributions are as follows.

- Introducing the first efficient, practicable, and scalable methodology for privacy-preserving k-NNS assuming that the dataset and query are each privately held.
- Proposing a sequential circuit description for privacy-preserving k-NNS using Yao's Garbled Circuit protocol (instead of the known combinational representation). New transformations are created such that the sequential k-NNS implementation are securely evaluated by interfacing with the available (combinational) cryptographic garbling schemes.
- Reduction in the size of the required memory for GC from $O(nw)$ to $O(w)$ compared with the best known GC implementation of 1-NN [168]. Our scalable implementation requires a memory in the order of $O(kw)$ for k-NNS. Note that k-NNS was impracticable earlier (for large n) due to the linear growth of the combinational representation.
- Proof-of-concept implementation of privacy preserving k-NNS utilizing the on an Intel processor. For example, the circuit size for k-NNS with $w = 31, k = 8$ is only 80KB.

6.4.2 Related Work

The related literature in realizing privacy-preserving k-NNS has mainly focused on using homomorphic encryption as the enabling cryptographic primitive [169, 170]. In their protocol, two parties perform k-NNS locally on their respective private dataset for a public query and then privately combine their results to form the k-NNS. In contrast with these works, we adopt a more general setting in which one party holds a private dataset and the other one provides a private query. The use of GC for privacy preserving data mining has been suggested, but the existing literature focused on theoretical/protocol aspects and not implementation [171]. Leveraging our sequential description, this paper proposes the first scalable implementation and a low-overhead realization of secure k-NNS on a conventional processor.

6.4.3 Distance Function

For k-NNS in 2D space, the default distance function would be Euclidean distance, which computes the length of the straight line path between the two points. However, in practice there is almost never a straight line path between two cars on the road as demonstrated with an example in Figure 6.9. We employ a more practical and computationally efficient *taxicab distance*. The taxicab distance, d_t between two points with rectangular coordinates (x1, y1) and (x2, y2) is given as $d_t = (|x1 - x2| + |y1 - y2|)$ As evident from the example in Figure 6.9, this distance function closely resembles the actual distance the driver has to cover to reach the rider. Note that, the k-NNS presented here is compatible with any distance function. For example, in the generic k-NNS presented in [20], the distance function was Hamming distance.

6.4.4 Generation of Netlist

As already mentioned, all the circuits are synthesized using the methodology presented in Section 3.2 To realize the k-NNS, a set of basic arithmetic and conditional operations consisting of comparator, multiplexer, and distance function are required. We create a custom synthesis library that includes the minimum non-XOR implementations of these operations. A w-bit

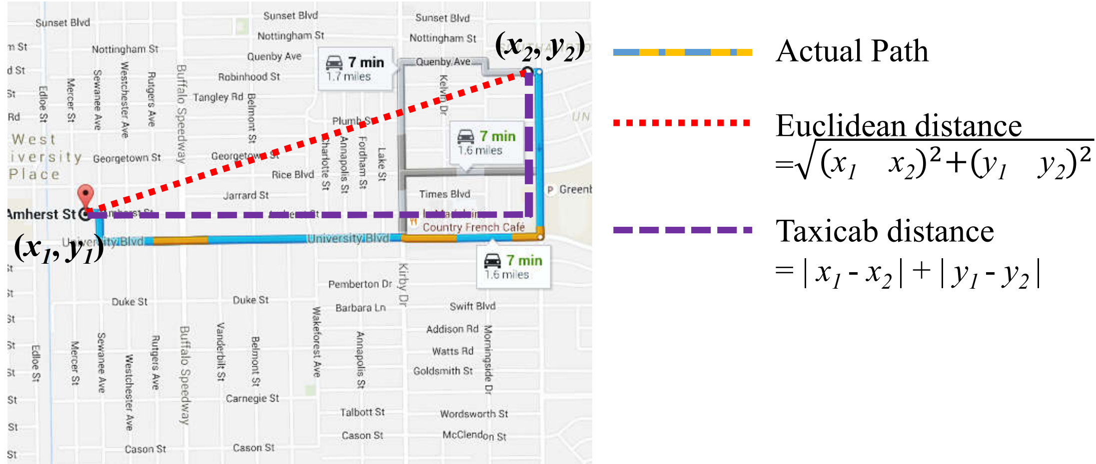


Figure 6.9. Illustration of the actual path, Euclidean distance and taxicab distance

comparator ($COMP_w$) is implemented using only w non-XOR gates [168]. A w-bit multiplexer (MUX_w) is realized using w non-XOR gates [26]. A w-bit taxicab distance (TD_w) is devised using $7w+1$ non-XOR gates. In all these modules, the total number of gates is $O(w)$.

6.4.5 Combinational Garbled Circuit

Prior tow our work on k-NNS, all previous implementations of use a combinational description. To start our implementation for the special case of 1-NNS, we look for the closest point (o) to the query point (q) in the dataset (S). In the privacy-preserving setting, there is a need to compare the query point to all the points in the dataset. This is because the (private) intermediate search values cannot be utilized to bound the search, e.g., binary search.

Figure 6.10 shows the combinational circuit for 1-NNS. The implementation uses n taxicab distance modules, and $(n-1)$ *min* modules (consisting of 1 COMP and 2 MUXs) to find the nearest point. One MUX selects the smaller distance for later comparison while the other one finds the point corresponding to that distance. The total number of gates in the 1-NNS

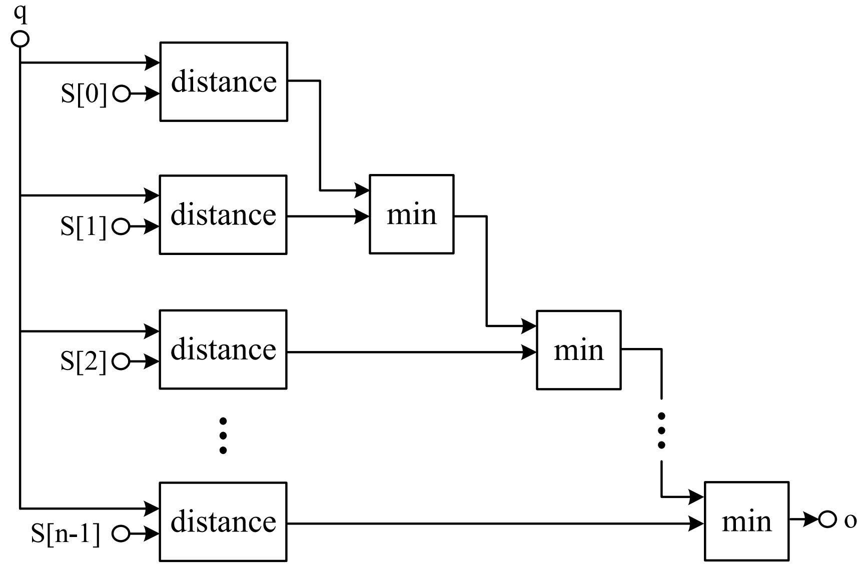


Figure 6.10. Combinational circuit for 1-NN. It consists of n taxicab distance and $(n-1)$ min modules.

combinational circuit is as follows.

$$\begin{aligned} \text{\# of gates} &= n \times TD_w + (n-1) \times (COMP_{w+1} + 2MUX_{w+1}) \\ \Rightarrow \text{\# of gates} &\in O(nw). \end{aligned}$$

The circuit should be garbled/evaluated only once. Thus, the time complexities of garbling/evaluation is $O(nw)$.

6.4.6 Sequential Garbled Circuit

Sequential circuits can be used as a very compact circuit description for both real hardware and GC protocol. A sequential circuit is composed of a combinational circuit and a set of registers that stores the intermediate values. We modify the garbling scheme such that for each sequential cycle, it garbles/evaluates the combinational part and stores the garbling keys for the registers. The stored keys are used as inputs in the next cycle. To ensure security, each gate should have a unique identifier for each time that it is garbled/evaluated. Since in the sequential circuit each gate is garbled/evaluated multiple times, we use the combination of gate index and cycle index as

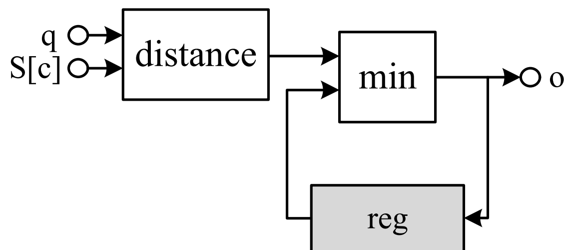


Figure 6.11. Sequential circuit for 1-NNS. It consists of 1 taxicab distance and 1 min module. For a dataset of size n, the circuit is required to be garbled/evaluated n times.

a unique identifier for each gate invocation. Thereby, the proof of security provided in [172, 21] also applies to our garbling scheme. We now describe the sequential 1-NNS implementation followed by k-NNSimplementation.

Sequential 1-NNS. Our 1-NNS sequential circuit is implemented with only 1 taxicab distance and 1 min module. Figure 6.11 illustrates the sequential circuit for 1-NNS. In each cycle c, the circuit computes the distance between q and $S[c]$. Next, it compares the resulting distance with the stored minimum distance in the register (reg). It then stores the minimum distance along with the nearest point until cycle c. The total number of cycles required to compute 1-NNS is n.

The total number of gates in the 1-NNS sequential circuit is as follows:

$$\begin{aligned} &\text{\# of gates} = TD_w + COMP_{w+1} + 2MUX_{w+1} \\ \Rightarrow &\text{\# of gates} \in O(w). \end{aligned}$$

The circuit should be garbled/evaluated n times. Thus, the time complexities of garbling/evaluation are the same as the combinational circuit and equal to $O(nw)$.

Sequential k-NNS. In k-NNS, the goal is to find the k nearest points to the query in the dataset. We expand the sequential circuit for the 1-NNS to store the k nearest points. For this purpose, we implement a priority queue with depth of k which receives one point at each cycle. The priority of each point is equal to its distance to the query. Figure 6.12 shows the sequential circuit for the k-NNS. The circuit has 1 taxicab distance, k min, and $k-1$ max modules. The

max module, like min, consists of 1 COMP and 2 MUXs.

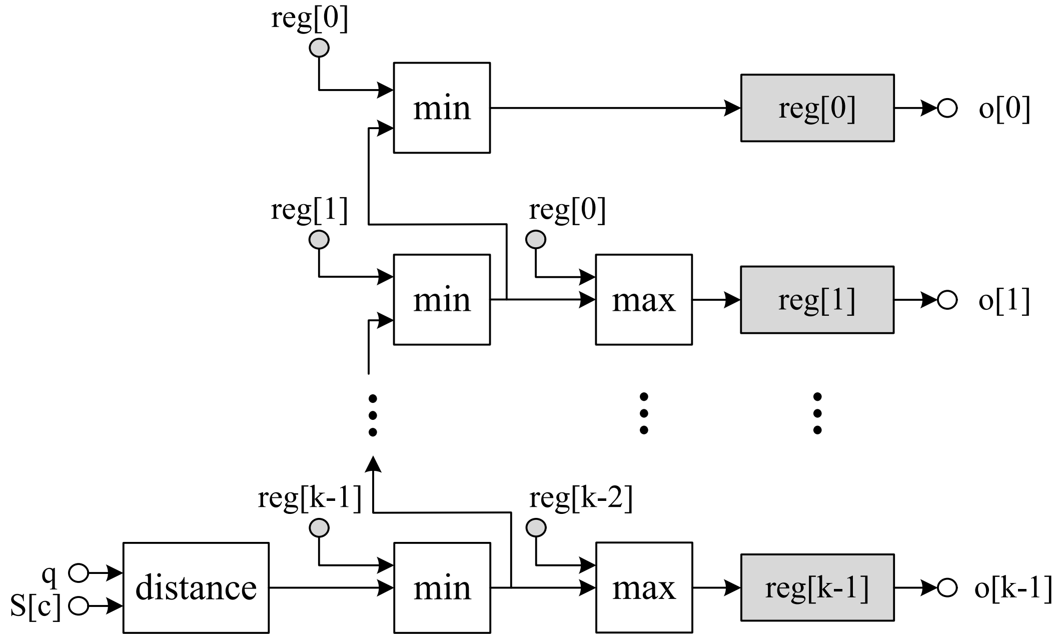


Figure 6.12. Sequential circuit for k-NNS. It consists of 1 taxicab distance, k min, and $k-1$ max modules. It requires to be evaluated n times where n is the size of the dataset S.

The total number of gates in the 1-NNS sequential circuit is as follows:

$$\text{\# of nonXORs} = TD_w + (2k-1) \times COMP_{w+1} + 2(2k-1) \times MUX_{w+1}$$

$$\Rightarrow \text{\# of nonXORs} \in O(kw).$$

The circuit should be garbled/evaluated n times. Thus, the time complexity of garbling/evaluation is equal to $O(nkw)$. Note that due to the unscalability of combinational k-NNS, we did not include its implementation.

6.4.7 1-NNS in Multi-Party Setting

In our first implementation of k-NNS, we adopted a two-party setting and employed the GC protocol. In our subsequent work MPCircuits [9], we extended this work to the multi-party setting. In this setting, there is now centralized server holding the location of all the available drivers. Instead, each driver participate in the computation with his location as the input. Note that

the MPCircuits framework supports efficient netlist generation but does not provide scalability. As a result, we were only able to implement 1-NNS due to the issues presented in Section 6.4.5.

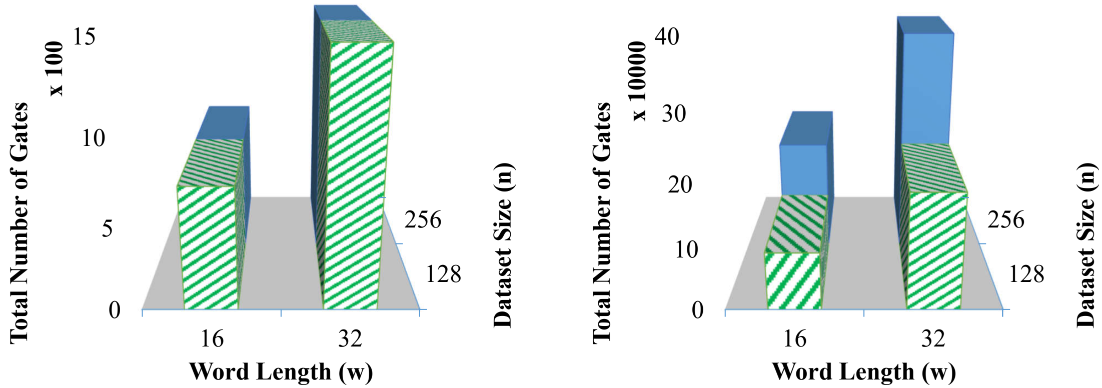


Figure 6.13. Comparison of memory footprints of 1NNS with combinational and sequential approach

6.4.8 Evaluation: Memory Footprint of 1-NNS

We compare our approach with previous ones for the special case of $k = 1$ since higher values of k were prohibitive with the previous approach. The memory footprint (circuit size) depends on the total number of gates in the circuit. Figure 6.13 shows the total number of gates as a function of the input word length, w and library size, n. We observe that while the memory footprint increases linearly with n for combinational approach, it is independent of n for sequential approach. Moreover, the circuit size is orders of magnitude smaller with our approach.

6.4.9 Evaluation: Timing of 1-NNS

The garbling/evaluation time is proportional to the total number of non-XOR gates that needs to be garbled. Theoretically, garbling time for both combinational and sequential approach should be similar. However, as shown in Figure 6.14 the computation time is reduced with sequential approach. This has two reasons. First, with reduction in circuit size, optimization by the logic synthesis tools is more effective resulting in reduction in the number of non-XOR gates. Second, with lower memory footprint for sequential circuit, there are fewer cache misses

resulting in faster operation.

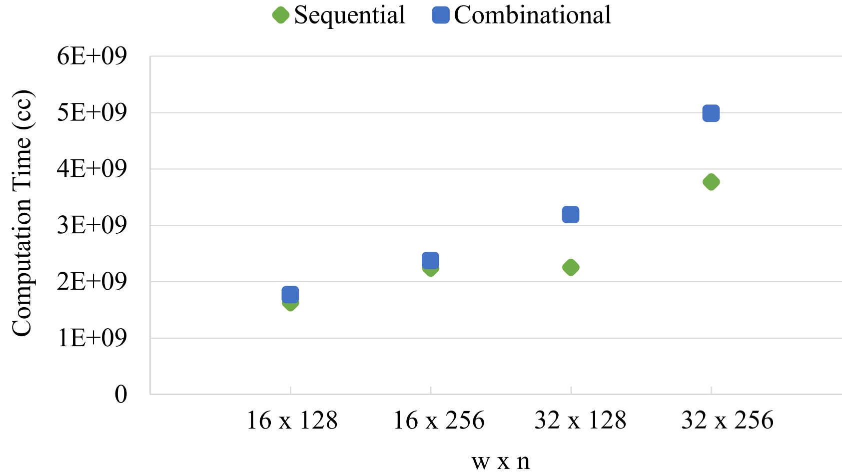


Figure 6.14. Comparison of garbling times of 1NNS with combinational and sequential approach

6.4.10 Evaluation: Memory Footprint of k-NNS

Figure 6.15 shows the total number of gates in sequential k-NNS circuit as a function of the input word length, W and k. As expected, it increases linearly with both W and k. As already explained, the total number of gates is independent of the library size, N.

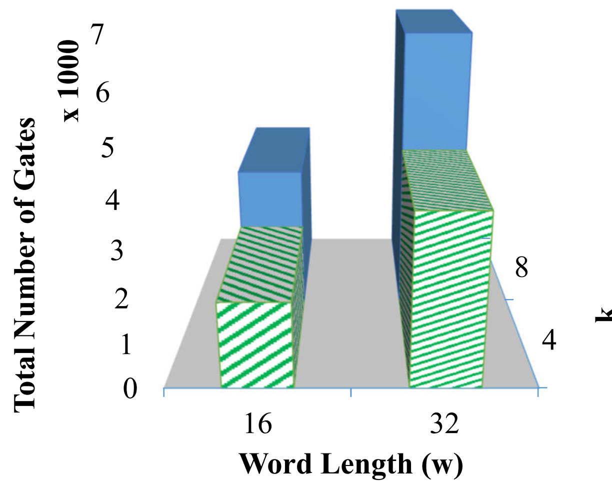


Figure 6.15. Memory footprint of k-NNS with sequential approach

The actual memory footprint for the largest circuit in this work ($w = 31, k = 8$) is 80KB which will fit easily in an embedded systems.

6.5 Private Set Intersection

Private Set Intersection (PSI) allows two or multiple parties to obtain the elements at the intersection of their sets without revealing the other elements that are not in common. For example, multiple people can identify their mutual contact profiles/friends by inputting their contact list to the PSI protocol without revealing the rest of their contact lists. At the end of the protocol, only the mutual list of all parties is revealed.

In E-commerce, an online advertisement agency and a company can participate in the PSI protocol where the advertisement agency inputs its list of all the people who have been shown the ads of the company. The second set of inputs to the protocol is the list of the people who have bought the products provided by the company. At the end of the PSI protocol, both entities know how many people have bought the product as a result of seeing the advertisement. This provides a way to understand the effectiveness of the advertisement for the company. Note that the same process could not be realized in plaintext due to various privacy/security reasons. Revealing such information is privacy invasive and can damage the reputation of both the companies. In addition, disclosing customer's data might be against the law in some situations.

In our setting, party P_i holds a set $S_i \subset \Omega$ where Ω is the universal set. Together the parties compute the intersection set $S = \cap_{i=1}^{n} S_i$. The size of the universal set Ω or equivalently the number of bits required to describe an element in the universal set is $b = \lg|\Omega|$. Maximum number of elements in each party's set is m.

6.5.1 Circuit Design

We designed the circuits required for PSI using the MPCircuits framework. Two different implementations are provided for PSI: a Bitwise-AND based circuit and a Sort-Merge-Compare-Shuffle (SMCS) based circuit. The first one is more efficient for scenarios in which Ω is small whereas the second approach is more suitable when m is small and Ω can be very large. Note that sets are represented differently in the two implementations as we explain in each section.

Bitwise-AND. In this implementation, each set is equivalent to a binary vector. The binary value at index j denotes the presence of the j-th element in a given set. Therefore, each set is represented as a $|\Omega|$-bit binary vector. The intersection set S is computed as *bit-wise* AND between all of the sets provided by all parties. As a result, the complexity of the circuit is $O(n\,|\Omega|)$, linear in both the number of parties and the size of the universal set; but *independent* from the number of elements in each parties' set m.

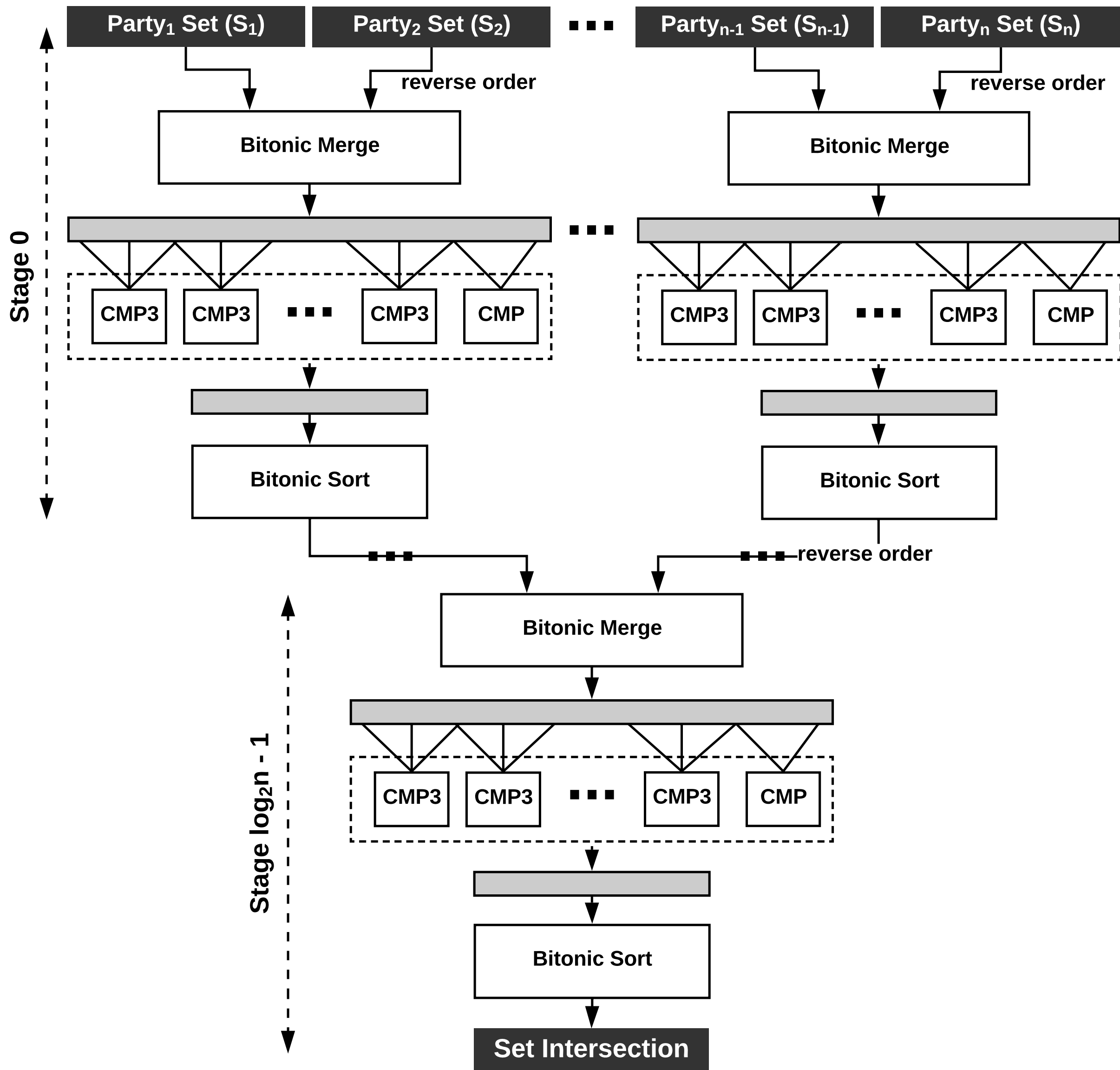


Figure 6.16. High-level circuit description of the Sort-Merge-Compare-Shuffle for Private Set Intersection. Three operations are performed at each stage: merge, compare, and sort.

Sort-Merge-Compare-Shuffle (SMCS). In scenarios where $m << |\Omega|$, more efficient solutions than Bitwise-AND can be devised. Here, we present one of the most complicated circuits in our benchmarks which is the generalization of the approach presented in [173] from two-party setting to any n-party case. As the input to this circuit, each set is represented as a vector of m integers where each integer is b-bit. We will first explain the solution for two sets only. The intersection of two sets can efficiently be computed using three operations: sort, merge, and compare. First, each of these two sets should be sorted. Then by merging the two sorted sets, all elements in common will be brought together. Finally, by comparing adjacent elements, one can find the common elements in both sets. Since the set intersection is an associative operation, one can express the set intersection of n sets as a consecutive set intersection of two sets until reaching the final result. Therefore, the SMCS circuit has a binary tree structure where at each node, the intersection of two sets are computed. The final node computes the final intersection of all sets. Note that the first sort operation can locally be computed by each participant since it is independent of the other parties' private data. A final shuffle operation is needed in order to eliminate the information leakage which we describe later in this section. Without loosing any generality, assume that the number of sets (participants) is a power of two. If this is not the case, dummy nodes can be avoided in the tree structure. Please see Figure 6.16 for a high-level description of the SMCS circuit.

We now elaborate on each part of the SMCS circuit. The challenge is that the merger and sorter circuits should have a *fixed* structure and *non-random access* to the intermediate values since random access is a very costly operation in the MPC protocols. We rely on the *bitonic* merger and sorter circuits that satisfy this condition. Bitonic sort is one of the sorting networks that is an efficient circuit-based realization of a sorting algorithm. Input numbers are given to the circuit and after series of conditional swap operations, a sorted list is given as the output of the circuit. The only operation used in the circuit is conditional swap: given two input numbers, swap them if they are not sorted and do not swap them otherwise. The bitonic sort has a recursive structure. It first sorts each half of the input and then merges the two sorted lists. The base case

is a circuit that sorts only two numbers which is equivalent to a conditional swap module. Our implementation of the bitonic sort circuit is also a recursive hardware description code.

The second half of the bitonic sorter represents the *bitonic merger* circuit. The input to the bitonic merger must be a bitonic sequence. A sequence x_i of numbers is called bitonic if for some k $(0 \leq k < m)$:

$$x_0 \leq x_1 \leq \ldots \leq x_k \geq \ldots \geq x_{m-1} \geq x_m$$

or a circular shift of such sequence. Therefore, before merging the two sorted lists, one needs to reverse order the second list such that the concatenation of two lists be a bitonic sequence. This reverse-ordering should take place for input sets as well as for intermediate sets. Note that the reversing the order of a set does not incur any computation or communication cost and is realized as changing the order of wires in the circuit.

The second layer in the SMCS circuit is the *comparison* layer. After the merger layer, all identical elements in both sets are now beside each other. An intuitive solution is to have a series of comparison blocks that compare every two adjacent elements. However, it has been shown that having a 3-input comparison block as follows is more efficient [173]:

$$\text{CMP3}\ (x_1, x_2, x_3) = \begin{cases} x_2 \ \ if\ x_1 = x_2 \mid x_2 = x_3 \\ 0^b \ \ otherwise \end{cases}$$

Given an array of $2m$ elements, we only need $m-1$ CMP3 blocks and one CMP block (compared to $2m$ CMP blocks).

The output of the comparison layer is an array of m numbers consisting of 0^b and the elements in the intersection of two sets. Before proceeding to the next stage (and similar to the first stage), the array has to be sorted. Note that the intermediate sets should not be revealed to any party since some information about the private input sets will be learned by other parties. Therefore, in contrast to the first stage, the sets should be sorted inside the MPC protocol.

At the end of all stages, the final set should be shuffled prior to be revealed in plaintext to

all parties. This step is necessary because the final set potentially has a sequence of 0^b between two common elements. The position of zeros (0^b) reveal the distribution of elements that were not in the intersection and belong to one (or multiple parties) only.

The shuffling layer can be realized using Waksman permutation network [174] which takes as input an array and shuffles them based on the *control bits*. One of the parties is required to provide these control bits as well. However, this task gives one of the parties more control in the secure computation. For example, a dishonest party that is selected to provide the control bits can simply put all of them as zero which makes the shuffle layer ineffective and he can learn some information. As a result, we devise another solution that is secure but does not require more input from any party. The solution is to simply sort the final list before revealing it in plaintext. This approach is secure since all of the 0^b elements are brought together. More precisely, in all of the scenarios that the common elements are fixed, the final sorted set remains the same and an adversary cannot distinguish different scenarios. Overall complexity of SMCS circuit is $O\big(n\,m\,\lg^2 m\,b\big) = O\big(n\,m\,\lg^2 m\,\lg|\Omega|\big)$ (compare with Bitwise-AND circuit with complexity $O\big(n\,|\Omega|\big)$).

Modular Structure. One of the advantages of using a generic secure multi-party computation protocols such as BMR is its modular nature and flexibility. Unlike customized protocols, additional functionalities and computations can be augmented to the circuit seamlessly. For example, and auditing step can be added before releasing the final result: the intersection set is revealed if and only if the number of elements in common is less than a threshold. Such auditing steps are favorable especially when Ω is small and an adversary can easily put his input set as the universal set in which case, he clearly learns the intersection of all other sets. As another example, it is very straightforward to build other variants of PSI such as PSI-Cardinality which only outputs the size of the intersection and not the elements.

6.5.2 Evaluation

Table 6.7 shows the experimental results of Bitwise-AND circuit for different sizes of the universal set and different numbers of parties. For all PSI experiments, parameter m is set to 16. The corresponding results for the SMCS circuit are shown in Table 6.8. As can be seen, the optimized Boolean circuits using MPCircuits technology libraries reduce the number of AND gates by 4.2×.

There has been extensive research focused on PSI for a two-party situation [175, 176, 173]. In [173], authors propose a method for two-party PSI based on garbled-circuit approach. To the best of our knowledge, the only solution that is proposed for secure multi-party private set intersection is a recent work by Kolesnikov et al. [177]. Their present a customized solution optimized only to perform PSI in an identical security model as this work. Their computation platform is comparable but more powerful than ours. In the LAN setting, for a set size of 2^{16} and 10 parties, their total running time is 12 seconds with 23MB of communication. Whereas, for a universal set of size 10^5 ($\sim 2^{17}$) and 8 number of parties, our running time is 24 seconds with 314MB of communication. Although our solution is less optimized, we want to emphasize that we have proposed a generic solution to create any functionality, whereas, their solution is specially optimized for PSI. In addition, our solution has a very modular structure and can easily be modified to support other variants of PSI, e.g., PSI cardinality in which only *number* of mutual elements is revealed. Moreover, in Bitwise-AND circuit, actual size of each party's set is not revealed since the inputs are fixed-length binary vectors.

Table 6.7. Private set intersection (Bitwise-AND variant).

		Non-optimized		**Optimized**						
$\|\Omega\|$	n	#XOR	#AND	#XOR	#AND	OT (s)	T_{GE} (s)	T (s)	$Comm$ (MB)	Mem (MB)
10^4	4	0	3.00E+04	0	3.00E+04	0.94	0.69	3.89	12.36	65.24
	8	0	7.00E+04	0	7.00E+04	2.73	1.99	9.46	67.29	403.94
	16	0	1.50E+05	0	1.50E+05	12.61	4.74	30.46	308.99	2835.59
10^5	4	0	3.00E+05	0	3.00E+05	1.99	0.88	6.80	123.60	584.44
	8	0	7.00E+05	0	7.00E+05	11.82	2.89	24.05	672.91	3892.61

Table 6.8. Private set intersection (SMCS variant).

b	n	Non-optimized #XOR	Non-optimized #AND	Optimized #XOR	Optimized #AND	OT (s)	T_{GE} (s)	T (s)	$Comm$ (MB)	Mem (MB)
16	4	1.05E+04	7.77E+04	5.02E+04	1.86E+04	0.82	0.56	3.52	7.66	52.57
	8	2.42E+04	1.81E+05	1.16E+05	4.30E+04	2.42	1.76	6.59	41.37	280.42
	16	5.15E+04	3.88E+05	2.48E+05	9.19E+04	9.65	4.40	41.50	189.34	1843.72

6.6 Summary

In this section, we presented a number of real-world applications pertaining to privacy-sensitive data. These applications were developed based on the MPC frameworks presented in Chapter 3. The solutions presented in this chapter represents the state-of-the-art for the respective problems.

Acknowledgement. This chapter, in part, has been published at (i) 2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) and appeared as: Sadegh M Riazi, Mojan Javaheripi, Siam U Hussain, and Farinaz Koushanfar, “MPCircuits: Optimized Circuit Generation for Secure Multi-Party Computation”, and (ii) 2018 ACM Transactions on Design Automation of Electronic Systems (TODAES) and appeared as: Siam U Hussain, Sadegh M Riazi, and Farinaz Koushanfar, “SHAIP: Secure Hamming Distance for Authentication of Intrinsic PUFs”, and (iii) 2018 ACM Transactions on Design Automation of Electronic Systems (TODAES) and appeared as: Siam U Hussain, and Farinaz Koushanfar, “P3: Privacy Preserving Positioning for Smart Automotive Systems”, and (iv) 2016 ACM/IEEE Design Automation Conference (DAC) and appeared as: Siam U Hussain, and Farinaz Koushanfar, “Privacy Preserving Localization for Smart Automotive Systems”, and (v) 2015 ACM/IEEE Design Automation Conference (DAC) and appeared as: Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, and Farinaz Koushanfar, “Compacting Privacy-Preserving k-Nearest Neighbor Search Using Logic Synthesis”. The dissertation author was the primary investigator of (ii), (iii), and (iv).

Chapter 7

Co-optimization of Crypto Primitives and ML Inference

7.1 Overview

Recent algorithmic and technological breakthroughs in Machine Learning (ML) have led to a surge in cloud-based inference using Deep Neural Networks (DNNs). In this scenario, a server trains and holds the DNN model. Clients then send their data to the server to perform inference using the server's trained DNN. Cloud-based inference, a.k.a. Machine Learning as a Service (MLaaS), is integrated in a wide range of real-world applications such as personal assistants [178], face authentication [179], medical diagnosis [180, 181, 182, 183], and health monitoring [184]. However, plaintext DNN inference either violate the users' privacy by revealing their private data to the server or infringe the server's intellectual property by exposing its proprietary model/data to the client. This paper focuses on the critical subject of oblivious inference, where the server and the client participate in two-party secure computation to run inference without revealing either the model parameters or client's data.

We present COINN, a provably secure cryptographic framework that surpasses the efficiency of all known methods for oblivious inference to date. Our work addresses the tension between three critical requirements for privacy-preserving DNN inference, namely, security, efficiency, and accuracy. Although several prior works have attempted to solve this tri-objective, there still remains a large gap in the accuracy and/or runtime of oblivious inference and plaintext

DNN execution. To deliver a balanced tradeoff between the above three criteria, we co-design the DNN and the secure execution protocol and holistically optimize both aspects via our automated design configuration tool. Our key design goals are as follows:

❶ **Compact Communication and Computation:** We optimize the computation bitwidth to reduce the secure execution cost of both linear and nonlinear operations. In doing this optimization, we adapt techniques from Genetic Algorithms [185] to the constraints of secure computation. Moreover, we design efficient cryptographic protocols that reduce the communication cost of secure matrix-multiplication by 5×–9×, and achieve an end-to-end runtime speedup of 4.7×–14.4× over best prior work, namely CrypTFow2 [5], in the LAN setting.

❷ **Inference Accuracy:** COINN improves the accuracy of prior ML-security co-optimization methods, namely [3, 1, 2], by 0.6%–4.7% while achieving 23.1×–36.8× lower secure execution runtime in the LAN setting.

❸ **Scalability:** Our framework scales to DNNs with over 100 layers. COINN achieves 6.1×–7.8× lower runtime in the LAN setting for the largest ever studied image classification task [5] with over 4 billion arithmetic operations.

In what follows, we review the design challenges, survey the prior work, and specify our contributions in detail.

Security-aware Quantization. To reduce the high cost of ciphertext execution, contemporary methods modify the neural network architecture by removing/replacing non-linear operations such as ReLU [1, 186], or binarizing model parameters and activations [3]. While these methods increase the secure execution efficiency, they come at the cost of reduced inference accuracy. Our work approaches the problem from a different perspective. Since the computation and communication overheads of cryptographic protocols are highly dependent on the computation bitwidth, we focus on developing quantization methods that take into account the constraints of ciphertext computation.

Our low-bit quantization reduces the communication and computation cost for not

only linear but also nonlinear layers which are the main efficiency bottleneck reported in prior works [90, 187, 188, 1]. A critical design challenge is that off-the-shelf quantization methods used in the ML community comprise operations such as full-precision accumulation, rounding, and scaling; these operations are efficient in plaintext inference but require expensive cryptographic operations in ciphertext. To address this challenge, we devise a novel ciphertext-aware quantization scheme that replaces the costly operations with counterparts that are low-cost in the secure domain while minimally affecting the DNN accuracy.

A major concern in computation on quantized data is the linear growth of the computational bitwidth with increased multiplicative depth. To mitigate this growth, prior work [1] locally truncates the bits which sacrifices the correctness of ciphertext computation by introducing random noise as shown in followup work [5]. Developing cryptographic tools for truncation, as suggested in [189, 5], incurs additional secure execution cost. We address this issue at zero cost by simulating the effect of overflow in our ML quantization library. We further provide training methods compatible with our overflow simulation to fine-tune model weights and minimize the effect of overflow on model accuracy in the low-bit regime. This, in turn, eliminates the need for truncation altogether.

Efficient and Secure Linear Arithmetic. Matrix-multiplication comprises the core operation performed in linear layers of contemporary ML models. State-of-the-art oblivious inference frameworks employ either Arithmetic Sharing (AS) [187, 186] or Homomorphic Encryption (HE) [188, 1] for secure matrix-multiplication and Garbled Circuit (GC) for the nonlinear operations. In this work, we choose AS for efficient realization of secure linear layers since the secure conversion cost between AS and GC is $\sim 2.5\times$ smaller than the conversion cost between HE and GC[1]. Moreover, prior work [3] demonstrates that current HE-based methods [188, 1] would incur additional overhead to provide circuit privacy.

Our secure AS-based matrix-multiplication is optimized for the amortized setting, where

[1]The $\sim 2.5\times$ scale directly compares methods in Gazelle [188] (HE-GC) and ABY [32] (AS-GC). Further explanation is included in Section 7.5.3.

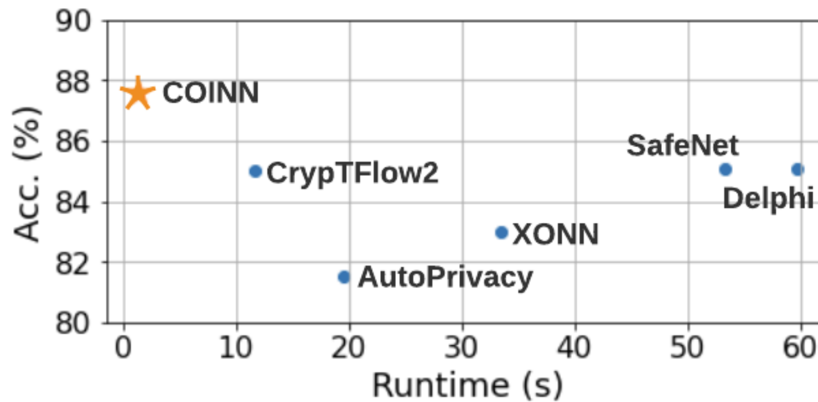


Figure 7.1. Accuracy and secure inference runtime of a 7-layer DNN on CIFAR-10 dataset using prior work: Delphi [1], SafeNet [2], XONN [3], AutoPrivacy [4], and CrypTFlow2 [5]. The ★ symbol represents COINN.

one client-server pair runs multiple inferences on the same trained model. In this setting, which is the common scenario in real-world applications, the matrix-multiplication in each linear layer is computed in a single round, thus reducing the effect of network latency on runtime.

Factored matrix-multiplication. We further optimize the linear layers and introduce repetition into the weight matrices. Our optimization ensures that only a limited set of unique values appear in each layer's weight matrix with minimal loss of inference accuracy. The unique values can then be leveraged to replace individual multiplications with *factored* ones. This, in turn, allows us to substitute the bulk of costly multiplications with cheaper conditional summations. Consider a dot product between two N-dimensional vectors, which requires N multiplications and additions. By ensuring that one vector contains V unique values, only N additions followed by V multiplications and additions are required. To accompany factored multiplication in cipher domain, we introduce an efficient custom protocol based on Oblivious Transfer (OT) that multiplies the factored weights with the activations without revealing either the unique values or their locations.

Automated Parameter Configuration. To fully exploit the efficiency gains from quantization and factored multiplication while minimally affecting the inference accuracy, COINN

automatically determines the best parameters for quantization and factorization across all DNN layers. By this cross-layer heterogeneous parameter selection, our system achieves a prominent advantage over previous work that use homogeneous and superfluous bitwidths [187, 190], as shown in Figure 7.1. The first challenge in finding the best set of per-layer parameters is simultaneous optimization of two of our objectives that are conflicting – accuracy and efficiency. To account for this tradeoff, we leverage a score function that captures both model accuracy and secure execution cost and assigns a quantitative measure of quality to each design configuration. The second challenge is the excessively large number of possible parameter configurations (search-space) that grows exponentially with model layers. We develop a highly scalable parameter optimizer based on genetic algorithms [185] to effectively traverse the large search-space. The score function is then used to guide our genetic algorithm to find the most optimal DNN for secure inference.

Binarized Neural Network (BNN). At one extreme, the factored matrix multiplication becomes equivalent to the Binarized Neural Network (BNN) [191] where the weights and activations are restricted to binary (i.e, ± 1) values. The benefits of employing BNNs for oblivious inference were first noted by XONN [3]. Despite achieving significant runtime improvement compared to the then state-of-the-art non-binary DNN inference, there are opportunities provided by BNNs that have not been leveraged by XONN. Part of the inefficiency of XONN is due to the usage of a single secure computation protocol (GC) as a blackbox for all neural network layers after the input layer. The bulk of the cost in XONN involved matrix multiplication through GC, which is significantly less efficient than AS for arithmetic operations. In this work, we customize our OT-based matrix multiplication protocol for efficient oblivious inference with BNNs. Moreover, we address the challenge of finding BNN architectures with both accuracy and oblivious inference efficiency by training a *single BNN* that can operate under different computational budgets. Our adaptive BNN offers a tradeoff between accuracy and inference time, without requiring to train separate models. With the combined power of our custom protocols

and adaptive BNN training schemes, our method outperforms prior art both in terms of accuracy and runtime. Our evaluation shows that we achieve 2× to 12× lower runtime compared to XONN.

COINN API. Our framework includes a high-level API that facilitates end-to-end deployment of user-defined DNNs for secure execution. Our API ensures that a user can employ COINN as a black box without knowing the details of underlying cryptographic protocols and DNN optimizations. The user provides the desired DNN model described in the well-known deep learning library PyTorch along with the trained model parameters. The custom-designed libraries of COINN for quantization and factored multiplication are then invoked through our automated design configurator to deliver the optimized DNN. Our framework also provides a seamless PyTorch interface to the secure inference engine developed in C++.

7.2 Related Work

In this section we present a brief overview of the related works on oblivious inference. We focus on the works that adopt the same scenario as ours, i.e., cryptographically secure two-party protocols in honest-but-curious setup where the server owns the model and the client owns the input. These works differ in the cryptographic primitives employed and their composition as well as their optimization domains. We divide the optimization approaches into two broad categories: (i) optimization of the cryptographic primitives to reduce the oblivious inference cost and (ii) optimization of the ML model to be better suited to the underlying cryptographic primitives. COINN is a bit unique in the sense that it optimizes the ML model and present new secure protocols to benefit from the new optimization. In the following we discuss different works in these two categories.

7.2.1 Cryptographic Optimization

There are two classes of techniques: Homomorphic Encryption (HE) [192], which is heavy on computation and Multi-Party Computation (MPC) techniques such as Garbled

Circuits (GC) [7] and Arithmetic Sharing (AS) [14], which are heavy on communication. Earlier works on oblivious inference, e.g., CryptoNets [90] followed by a number of subsequent works [193, 194, 195, 196] employed HE as the cryptographic primitive. HE has the advantage of outsourcing majority of the computation to the server, which usually have better computational resource. However, frameworks that are entirely based on HE replace the nonlinear activations with HE-friendly polynomial approximations, resulting in reduced inference accuracy. GC-based oblivious inference has been proposed by DeepSecure [197], and the work in [198]. These works provide better accuracy compared to HE based methods but usually suffers from long runtime due to high communication cost of GC.

Following the work in MiniONN [187], current oblivious inference frameworks employ a hybrid approach where the most efficient cryptographic primitive is employed for a particular task. MiniONN employed AS for the linear layers and GC for the non-linear layers. Later Gazelle [188] presented a hybrid framework with HE-based linear layers. Recently, CrypTFlow2 [5] proposed a hybrid framework with custom protocols for the non-linear layers. Their protocol resembles GMW which incur less communication at the cost of higher number of communication rounds compared to GC.

7.2.2 ML Optimization

The works in this category focuses on applying optimizations to reduce the secure execution cost of previously proposed security protocols described above. There are two different directions – adjusting the parameters for the secure protocol and adjusting the DNN architecture. Examples of works in the first direction include [199, 4] that adjust the HE parameters for hybrid HE-GC protocols. In the second direction, XONN [3] presents a GC-based method based on Binarized Neural Networks (BNN) where all multiplications are replaced with cost-free XNOR operations. Delphi [1] and CryptoNAS [186] present optimization methods where part of the non-linear activations are replaced with HE-friendly approximations with without hurting the accuracy much. Nevertheless, both binarization and approximate non-linear layers result in

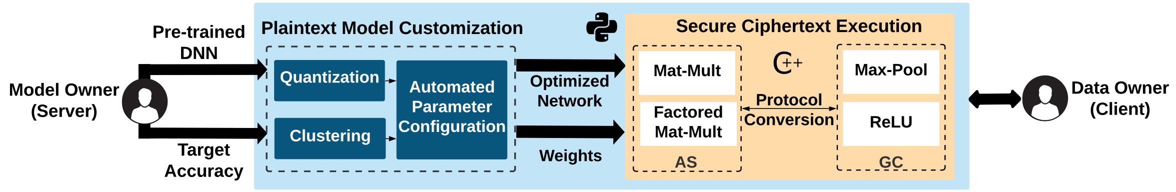


Figure 7.2. Overview of COINN. The plaintext model customization is only performed once per DNN and provides the optimized network for COINN secure inference.

reduced inference accuracy.

7.3 Global Flow and Threat Model

The COINN framework is composed of two interlinked components as depicted in Figure 7.2: (i) model customization on plaintext training data and (ii) secure execution on client's private input. We use the PyTorch library to describe the DNNs and develop our secure execution protocols in C++. In the following, we briefly introduce the incorporated design units.

Plaintext Model Customization. This is a one-time pre-processing performed on pre-trained full-precision DNNs prior to oblivious inference. Plaintext model customization is an important contributor to COINN efficiency and scalability as it enables customization of any given DNN for minimized secure execution cost under an accuracy constraint. Section 7.4 encloses the details of our plaintext model customization and its core components, i.e., cipher-text aware quantization, factored matrix-multiplication, and automated parameter configuration.

Secure Ciphertext Execution. We perform the linear operations such as CONV, FC through AS, and the nonlinear operations such as ReLU, MP through GC. Wherever necessary, we securely convert between AS and GC. We devise efficient cryptographic protocols that complement our optimized DNN models in the ciphertext domain. Our cryptographic components benefit from low-bit quantization performed by our model customization step. We also develop efficient AS-based protocols for both regular and factored matrix-multiplications. A thorough explanation of our end-to-end oblivious inference and cryptographic protocols is provided in Section 7.5.

7.3.1 Threat Model

COINN presents privacy-preserving protocols involving two parties: Alice – the server, and Bob – the client. The private inputs of Alice and Bob are trained weight parameters of the DNN and input to the DNN, respectively. At the end of the protocol execution, Bob learns the inference results without revealing any information to Alice. Following previous works on privacy-preserving neural network inference [188, 187, 5], we adopt an honest-but-curious security model where the two parties follow the agreed upon protocol, yet may try to learn more from the information at hand.

Consistent with prior work, we assume the information related to model architecture is public to the client and the server. This information includes number of layers, layer types, layer dimensions, number of bits required to represent the output of nonlinear layers, and AS computation ring size $\mathbb{Z}_{2^b}$. In our factored matrix multiplication (Section 7.4.2), the client additionally knows the per-layer number of unique weight values V but he is not aware of the distribution of the unique values inside the weight matrices.

Most prior works assume a large bitwidth b across all DNN layers (e.g., 32-bit ring size and activations in [5]). In contrast, COINN uses smaller bitwidths, e.g., it may use $b = 16$ for the ring size and $b = 10$ for the output of nonlinear layers. Exposing the customized b at each layer might reveal some information about the context and/or distribution of the training dataset. It is unclear whether this information can give additional advantage to an attacker. In addition, having lower bitwidths may reduce the computational complexity for extracting the neural network weights. Let us consider an attacker who launches a brute-force attack without any prior knowledge. The computation complexity for such an attack is $O(2^{bn})$, where n is in the order of millions. Therefore, even with the minimum bitwidth ($b = 1$ as in XONN [3]) the attack complexity, i.e., $O(2^n)$, is still exponential in n. Similarly, by knowing the unique size V for factored matrix multiplication, the attack complexity is $O(V^n)$.

A more knowledgeable adversary might try to employ more sophisticated attacks such as model extraction [200], model inversion [201], and membership inference [202]. Similar to related work in oblivious inference [187, 188, 3, 5, 1, 186], COINN does not address these query-based attack algorithms. Mitigating such attacks is also an active area of research and in most cases is orthogonal to our work [203, 204, 205, 206, 3]. Example mitigation strategies include differential privacy, rounding the prediction vector, or returning only the argmax of the prediction to the client. We refer the curious reader to [3]-Appendix B and [1]-Section 8.2 for more discussions.

7.4 COINN Model Customization

In the proposed system, the training is performed locally by the server in plain-text. We customize the model during training process to reduce the cost of secure inference. In this section we present the details of the model customization.

7.4.1 Ciphertext-aware Quantization

One crucial early step in secure inference is quantization of the weights and activations to integers. Even though ML libraries generally represent data with 32-bit floating-point format (`FP32`), the extremely high computational cost and complex circuits make `FP32` unsuitable for secure computation. In what follows, we discuss the challenges of such quantization with regards to secure computation and how we tackle them.

Optimizing Scaling. `FP32` values are converted to fixed-point by multiplying them with appropriate scale s, the value of which depends on the range of the all the values in a particular ML model. On one hand, such multiplication during the AS-based matrix-multiplication would result in increased multiplicative depth. This would in turn increase AS computation bitwidth thereby sacrificing the overall efficiency. On the other hand, if scaling is performed in the GC domain at the end of the matrix multiplication, for a b-bit number with a scale containing b'

nonzero bits, the communication cost would be $2b(b'-1)\times\kappa$. In COINN, to reduce the cost, we enforce the scale values to be powers of 2 ($b' = 1$), which allows us to implement the previously costly scale operation with almost zero cost logical shifts in GC.

Rounding Workaround. Rounding a fixed-point value requires adding the first fractional bit to the integer. The GC cost of this operation is $2b\times\kappa$, which is quite significant considering it has to be repeated for all output elements across all DNN layers. In COINN, we replace $round(\cdot)$ with the floor operation $\lfloor\cdot\rfloor$ and fine-tune the model weights to adjust to this modification. Flooring is equivalent to removing all fractional bits and therefore incurs no GC cost.

Overflow Management. An imminent challenge when performing matrix-multiplication on quantized values is managing overflow in the accumulator. Existing plain-text methods handle overflows with the $clamp(\cdot)$ operation which maps overflown values to the maximum/minimum valid range. Secure implementation of $clamp(\cdot)$ through GC incurs a communication cost of $2(3b+1)\times\kappa$ for each multiplication and addition, Moreover, it requires repetitive conversion between AS and GC resulting in a significantly high cumulative cost. COINN presents the first ML library that simulates overflow for quantized operations and supports overflow-aware training. It thus allows us to adjust the weights and quantization bitwidths such that the adverse effect of overflow on inference accuracy is minimized.

7.4.2 Factored Matrix-Multiplication

Factored Matrix-Multiplication replaces the majority of costly multiplications with cheaper conditional addition. Consider a matrix-multiplication of the form $Y = W\cdot X$, where $Y\in\mathbb{R}^{M\times L}$, $W\in\mathbb{R}^{M\times N}$, and $X\in\mathbb{R}^{N\times L}$. This operation requires $M\times L$ VDPs on vectors of length N, $\mathbf{w}\in\mathbb{R}^N$ and $\mathbf{x}\in\mathbb{R}^N$, corresponding to a row of W and a column of X, respectively. Each VDP therefore requires N multiplications and N additions. We introduce the factored VDP which forms the core of factored matrix-multiplication. We start with the definitions of the unique space and the coded representation of vectors involved in VDP.

Definition 1. *The unique space of* $\mathbf{w} \in \mathbb{R}^N$ *is the set* $\mathbf{c} = \{c_1, \ldots, c_V\}$ *such that* $\mathbf{w}[i] \in \mathbf{c}$ $(\forall\, i \in [N])$. *We refer to* V *as the unique size of* $\mathbf{w}$.

Definition 2. *Given a vector* $\mathbf{w} \in \mathbb{R}^N$ *and its unique space* $\mathbf{c} = \{c_1, \ldots, c_V\}$*, the coded representation of* $\mathbf{w}$ *is a vector of integer indices* $\widetilde{\mathbf{w}} \in [V]^N$ *such that* $\mathbf{w}[i] = \mathbf{c}[\widetilde{\mathbf{w}}[i]]$.

Knowing the unique space $\mathbf{c}$ and the coded representation $\widetilde{\mathbf{w}}$, the VDP is computed as follows. We first compute N conditional additions, each of which adds an input vector element to one of V accumulators based on its code:

$$\mathbf{s}[v] = \sum_{x \in \mathbb{S}_v} x, \qquad \mathbb{S}_v = \{\mathbf{x}[i] \mid \widetilde{\mathbf{w}}[i] = v\} \tag{7.1}$$

Next, a VDP is computed between the accumulated values and the unique space of $\mathbf{w}$. Factored VDP requires V multiplications and $N+V$ additions. Its benefits are most substantial when $V << N$[2]. In general, V can be as large as 2^b, where b is the quantization bitwidth of $\mathbf{w}$. Even with lower bitwidths of $\mathbf{w}$, V can be quite large, e.g., $V = 64$ for 6-bit weights. We perform clustering [207] on the weight matrices with several representative elements to decrease V. Clustering is performed by the server over plaintext weight matrices to obtain $\mathbf{c}$ and $\widetilde{\mathbf{w}}$ which are then used in ciphertext execution (Section 7.5.1).

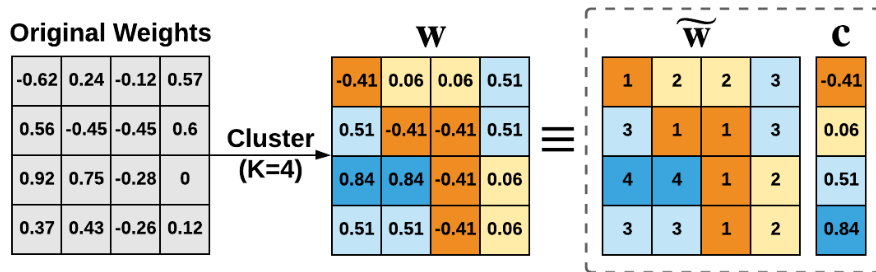


Figure 7.3. Example 4×4 weight matrix approximated via clustering with $V = 4$. The approximated matrix W can be represented as a tuple $(C, \widetilde{W})$.

---

[2] N is in the order of 100-10000

Table 7.1. TytaNN secure execution cost for core operations in a DNN. Here, κ is the security parameter that is set to 128.

	$\textbf{input}_{\textbf{dim}} \xrightarrow{\textbf{operation}} \textbf{output}_{\textbf{dim}}$	**Ciphertext Cost**		**Parameters**
Mat-Mult	$X_{N\times L} \xrightarrow{W_{M\times N}} Y_{M\times L}$	regular	$O\big(M\cdot N\cdot L\cdot b_{acc}(b_{acc}+1)\big)$	b_{acc}: accumulator bitwidth
		factored	$O\big(M\cdot K\cdot L\cdot b_{acc}(n+b_{acc}+1)\big)$	K: unique size of W
MaxPool	$X_{C\times D_1\times D_1} \xrightarrow{k\times k} Y_{C\times D_2\times D_2}$	$O\big(4\kappa\cdot C\cdot D_2\cdot D_2(k^2-1)b_{inp}\big)$		b_{inp}: next layer input bitwidth k: window size
ReLU	$X_{C\times D_1\times D_1} \xrightarrow{>0} Y_{C\times D_1\times D_1}$	$O\big(2\kappa\cdot C\cdot D_1\cdot D_1\cdot b_{inp}\big)$		b_{inp}: next layer input bitwidth
AS → GC	$X_{C\times D_1\times D_1} \xrightarrow{b_{acc}\to b_{inp}} Y_{C\times D_1\times D_1}$	$O\big(5\kappa\cdot C\cdot D_1\cdot D_1\cdot b_{acc}\big)$		b_{acc}: accumulator bitwidth
GC → AS	$X_{C\times D_1\times D_1} \xrightarrow{b_{inp}\to b_{acc}} Y_{C\times D_1\times D_1}$	$O\big(3\kappa\cdot C\cdot D_1\cdot D_1\cdot b_{inp}\big)$		b_{inp}: next layer input bitwidth b_{acc}: accumulator bitwidth

7.4.3 Automated Parameter Configuration

Clustering, as described in the previous section, is a form of lossy data compression, which if not carefully employed, may reduce the DNN inference accuracy [208]. Balancing trade-off between large unique space size V (thus low error) and low execution cost is a challenging task. V as well as the quantization bitwidths b across different layers are dependent on each other and they collectively control the tradeoff between model accuracy and secure execution cost. COINN is equipped with an automated parameter configurator that finds the minimum values of b and V across DNN layers such that the secure execution cost is minimized while accuracy is above a certain user-defined threshold.

Table 7.1 summarizes the communication cost associated with different ciphertext operations as part of oblivious inference by COINN. These cost are incorporated into our automated design customization tool. Note that for the linear layers, we report the amortized costs (see Section 7.5.2). For details of the configurator refer to [209].

7.5 Cryptographic Protocols

Figure 7.4 illustrates operations in plaintext DNNs and their corresponding secure computation in COINN framework. The linear layers – CONV and FC are executed through secure matrix-multiplication protocols in the AS domain. We provide protocols for both regular and factored matrix-multiplication for these two linear layers. We exploit the data

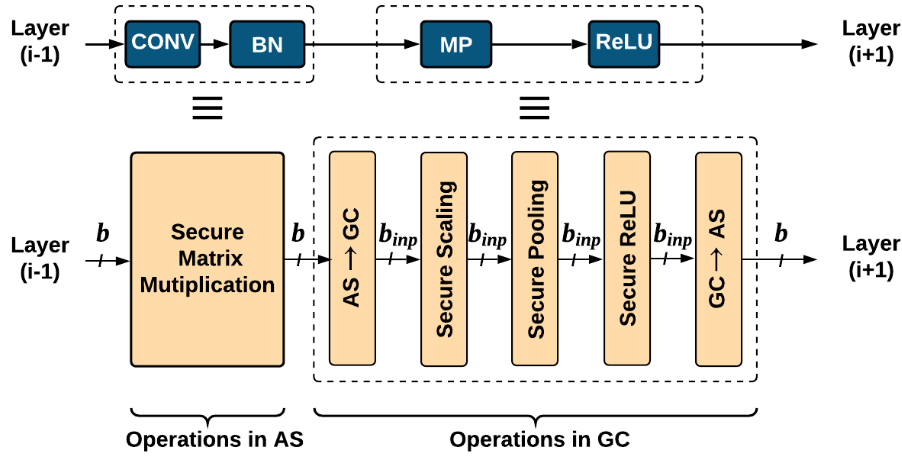


Figure 7.4. Plaintext operations and their equivalent ciphertext realization in COINN oblivious inference framework.

repetition inherent in the computation of matrix products to achieve a significant reduction in the communication cost. The outputs of the linear operations in the AS domain are securely converted to the GC domain for computation of the non-linear layers – MP and ReLU. The scaling operation is also embedded into the AS to GC (and vice versa) conversion. The design of these layers is optimized for the amortized setting where the same server-client pair runs multiple inferences without retraining the model, which is the common scenario in real-world applications.

Besides the aforementioned layers, COINN also supports BN and AP. These layers are fused into their preceding CONV/FC layers. Using this trick, heavy cryptographic operations such as the division protocol of CrypTFlow2 [5] can be avoided, allowing us to evaluate AP and BN at zero cost.

7.5.1 Matrix-Multiplication

As explained in Section 7.3, the weight matrix W is only known by Alice, i.e., $[\![W]\!] = [\![W]\!]_A$ and $[\![W]\!]_B = \mathbf{0}$ while the activation matrix $[\![X]\!]$ is shared between Alice and Bob. We need to compute the matrix product $[\![Y]\!] = [\![W]\!] \cdot [\![X]\!] = [\![W]\!]_A \cdot [\![X]\!]_A + [\![W]\!]_A \cdot [\![X]\!]_B$. Since Alice can locally compute $[\![W]\!]_A \cdot [\![X]\!]_A$, we focus on secure computation of $[\![Z]\!] = [\![W]\!]_A \cdot [\![X]\!]_B$.

Regular Matrix-Multiplication. The product $[\![Z]\!] \in \mathbb{Z}_{2^b}^{M \times L}$ of $[\![W]\!]_A \in \mathbb{Z}_{2^b}^{M \times N}$ and $[\![X]\!]_B \in$

$\mathbb{Z}_{2^b}^{N\times L}$ can be computed with MNL scalar multiplications. This approach requires $MNLb$ invocation of COT_b^2 [32], thus incurring a communication cost of $MNLb(\kappa+b)$ bits. The communication cost of one instance of COT_λ^2 is $\kappa+\lambda$, where the first term (the cost of one ROT_λ^2) is independent of the message bitwidth λ. Since the computation of the matrix product involves dot product of each row of $[\![W]\!]_A$ with L columns of $[\![X]\!]_B$, we compute the matrix product with MNb invocations of COT_{Lb}^2. This approach reduces the number of COTs by increasing the message length, thereby reducing the total communication cost to $MNb(\kappa+Lb)$. Compared to the protocols with independent multiplications [187, 32], the cost is reduced by a factor of $\frac{L(\kappa+b)}{\kappa+Lb}$. This cost can be further reduced in the amortized setting as will be explained in Section 7.5.2.

Factored Matrix-Multiplication. We now present our protocol for securely computing the factored matrix-multiplication explained in Section 7.4.2. We first define the one-hot encoded representation of a vector.

Definition 3. *Given a vector* $\mathbf{w} \in \mathbb{R}^N$ *and its coded representation* $\widetilde{\mathbf{w}} \in [V]^N$ *w.r.t. its unique space* $\mathbf{c} = \{c_1, \ldots, c_V\}$*, the one-hot encoded representation of* $\widetilde{\mathbf{w}}$ *is a matrix* $\widetilde{W} \in \{0,1\}^{V\times N}$ *such that* $\widetilde{W}[v,n] = 1$ *if* $\widetilde{w}[n] = v$ *and 0 otherwise* $(\forall v \in [V], n \in [N])$.

We will be using the following notations to explain our secure factored matrix-multiplication:

- Collection of unique spaces for all rows of $[\![W]\!]_A$: $\left\{[\![\mathbf{c}]\!]_A^{(m)} \in \mathbb{Z}_{2^b}^V\right\}_{m\in[M]}$
- Collection of one hot encodings of all rows of $[\![W]\!]_A$ w.r.t. $[\![\mathbf{c}]\!]_A^{(m)}$: $\left\{[\![\widetilde{W}]\!]_A^{(m)} \in \{0,1\}^{V\times N}\right\}_{m\in[M]}$
- Partial sum: $\left\{[\![S]\!]_A^{(m)} \in \mathbb{Z}_{2^b}^{V\times L}\right\}_{m\in[M]}$

Using the above notations, the product $[\![Z]\!] = [\![W]\!]_A \cdot [\![X]\!]_B$ is computed as:

$$[\![S]\!]_A^{(m)}[v,l] = \sum_{n=1}^{N} [\![\widetilde{W}]\!]_A^{(m)}[v,n] \cdot [\![X]\!]_B[n,l]; \forall m \in [M], v \in [V], l \in [L] \tag{7.2}$$

$$[\![Z]\!][m,l] = \sum_{v=1}^{V} [\![\mathbf{c}]\!]_A^{(m)}[v] \cdot [\![S]\!]^{(m)}[v,l]; \forall m \in [M], l \in [L] \tag{7.3}$$

Algorithm 2: Protocol for Computing Conditional Accumulation

Input : From Alice, one-hot encoding of weight matrix $[\![W]\!]_A$:
$\left\{[\![\widetilde{W}]\!]_A^{(m)} \in \{0,1\}^{V\times N}\right\}_{m\in[M]}$

Input : From Bob, share of the activation $[\![X]\!]$: $[\![X]\!]_B \in \mathbb{Z}_{2^b}^{N\times L}$:

Output : Partial sum $\left\{[\![S]\!]^{(m)} \in \mathbb{Z}_{2^b}^{V\times L}\right\}_{m\in[M]}$

OT message received by Alice, $\left\{\mu'^{(l)} \in \mathbb{Z}_{2^b}^{M\times V\times N}\right\}_{l\in[L]}$

OT message received by Bob, $\left\{\mu^{(l)} \in \mathbb{Z}_{2^b}^{M\times V\times N}\right\}_{l\in[L]}$

1 Bob chooses a set of correlation functions $\varphi_{m,v,n}^{(l)}(\cdot)$ as $\left\{\varphi_{m,v,n}^{(l)}(\mu^{(l)}[m,v,n])\right\}_{l\in[L]} = \left\{\mu^{(l)}[m,v,n] + [\![X]\!]_B[n,l]\right\}_{l\in[L]}; \forall m \in [M], v \in [V], n \in [N]$

2 **foreach** $m \in [M], v \in [V], n \in [N]$ **do**

3 Alice and Bob run COT_{Lb}^2 where

4 Bob acts as sender with correlation functions $\left\{\varphi_{m,v,n}^{(l)}(\cdot)\right\}_{l\in[L]}$ and receives $\left\{\mu^{(l)}[m,v,n]\right\}_{l\in[L]}$

5 Alice acts as receiver with choice bits $[\![\widetilde{W}]\!]_A^{(m)}[v,n]$ and receives $\left\{\mu'^{(l)}[m,v,n]\right\}_{l\in[L]} = \left\{\mu^{(l)}[m,v,n] + [\![\widetilde{W}]\!]_A^{(m)}[v,n] \cdot [\![X]\!]_B[n,l]\right\}_{l\in[L]}$

6 Alice sets $[\![S]\!]_A^{(m)}[v,l] = \sum_{n=1}^{N}\left(\mu'^{(l)}[m,v,n]\right); \forall m \in [M], v \in [V], l \in [L]$

7 Bob sets $[\![S]\!]_B^{(m)}[v,l] = \sum_{n=1}^{N}\left(\mu^{(l)}[m,v,n]\right); \forall m \in [M], v \in [V], l \in [L]$

Eq. 7.2 represents conditional accumulation and Eq. 7.3 represents dot product of length V vectors of b-bit integers. Note that the number of integer multiplications is reduced from MNL in regular matrix-multiplication to MVL in the factored version ($V \ll N$). The majority of the cost is now incurred by the conditional accumulation, which is computed through COT. We leverage our optimization presented for the regular matrix-multiplication, i.e., merging the COT messages involving the same selector bit, for both Eq. 7.2 and 7.3 to reduce the communication cost.

Algorithm 2 presents the protocol for computing the partial sums through conditional accumulation. Since the protocol requires MVN COT_{Lb}^2, the communication cost of computing the partial sums is $MVN(\kappa+Lb)$. The dot product of Eq. 7.3 can then be computed following the technique presented in MiniONN [187] with a communication cost of $MVb(\kappa+Lb)$. Thus the

total cost of computing factored matrix-multiplication is $MV(N+b)(\kappa+Lb)$. Since in practice, $b \ll N$, we set the cost to $MVN(\kappa+Lb)$ in the remaining discussion.

Proof Sketch. The security proof of Algorithm 2 directly follows from the security guarantee of OT. Observe that all the communication between Alice and Bob is performed through OT which ensures the privacy of both the selection bits and messages. Moreover, the correlation function chosen by Bob ensures that Alice never receives an unmasked version of any element of $[\![X]\!]_B$. Furthermore, every instance of OT involves freshly generated unique masks that ensures the security of the one-time pad.

7.5.2 Linear Layers in the Amortized Setting

The mean communication cost of both regular and factored matrix-multiplication is further reduced in the amortized setting where one server-client pair runs a large number of inferences with the same trained model but different inputs, i.e, W remains constant while X changes in each inference. In case of regular matrix-multiplication, since, W does not change, the number of COTs remains the same while message length increases to JLb, where J is the number of inferences. The mean cost per matrix-multiplication is therefore $MNb(\kappa+JLb)/J \approx MNLb^2$ for large J. Similarly, for factored matrix-multiplication, the mean amortized cost is $MVNLb$. More importantly, in this setting, the number of communication rounds remains constant (= 2), irrespective of the number of inferences J. Our protocol execution is split into setup, offline and online phases as described below.

Setup Phase. This is performed once per server-client pair irrespective of the number of inferences J. In this phase, for regular matrix-multiplication, Alice and Bob perform the MNb ROT^2_{JLb} as part of the MNb COT^2_{JLb} for matrix-multiplication computation. In practice, following the state-of-the-art OT libraries [35, 210], Alice receives MNb κ-bit seeds $\gamma_q; \forall q \in [MNb]$ and Bob receives κ-bit seeds γ_{0q} and $\gamma_{1q}; \forall q \in [MNb]$ which are later expanded to b-bit messages $\forall j \in [J], l \in [L]$ through Cryptographically Secure Pseudo Random Number

Generator (CS-PRNG). This makes sure that the memory requirement is independent of the number of inferences J. The communication cost of the setup phase is $MNb\kappa$. Similarly, the communication cost of the setup phase for factored matrix-multiplication is $MV(N+b)\kappa$.

Offline and Online Phases. These two phases are performed once per inference j. The offline and online phases involve computation *before* and *after* the input X is available, respectively. We employ the technique proposed by Slalom [211], to ensure that most of the cost corresponds to the offline phase. In this technique, in the offline phase, Alice and Bob securely compute the matrix product $[\![Z']\!] = [\![W]\!]_A \cdot [\![U]\!]_B$, where $[\![U]\!]_B \in \mathbb{Z}_{2^b}^{N\times L}$ is a random matrix generated and known by Bob. They locally expand the seeds obtained in the setup phase for the particular inference index j and for each column $l \in [L]$ of X and completes the COT^2_{Lb}. The communication cost of this phase for each $j \in [J]$ for regular and factored matrix-multiplications are $MNLb^2$ and $MVNLb$ respectively. In the online phase, Bob directly sends $F = [\![X]\!]_B - [\![U]\!]_B$ to Alice who locally computes $[\![Z]\!]_A = [\![Z]\!]_A + [\![W]\!]_A [\![F]\!]_A$. Bob sets $[\![Z]\!]_B = [\![Z']\!]_B$. The communication cost in this phase negligible compared to that of the offline phase.

Number of Communication Rounds. In the proposed setting, the only communication from Alice to Bob occurs in the setup phase. The offline and online phases involve communication from Bob to Alice, only. Thus the number of communication rounds is 2, irrespective of the number of inferences J. This reduces adverse effect of increased network latency in WAN setting.

Switching between Regular and Factored Multiplication. Based on the optimal unique size allocated to each layer's weights by the model configurator, our protocol automatically switches between regular and factored multiplication to maximize efficiency. Switching is done when the costs of both multiplications are equal, i.e., $MNLb^2 = MVNLb$ in the amortized setting, which renders the switching point $b = V$, i.e., when the number of unique values in each row of the weight matrix is equal to the bitwidth of the AS shares.

7.5.3 Non-linear Layers

COINN GC domain incorporates the following four stages:

(i) AS to GC Conversion. A variable $[\![x]\!] \in \mathbb{Z}_{2^b}$ shared between Alice and Bob through AS is securely converted to its share in the GC domain by securely computing the addition function through GC with inputs from Alice, the garbler and Bob, the evaluator as $[\![x]\!]_A$ and $[\![x]\!]_B$, respectively. Before the addition, Bob obtains the Yao share for his input $[\![x]\!]_B$ through COT, which requires two rounds of communication. In this particular scenario, Bob's share is generated through multiplication in the AS domain. According to the multiplication technique described in Section 2.6.1, his shares are independent of his input. Therefore, we perform the COTs for all the layers in parallel during the offline phase. This approach reduces one round of communication for each layer in the online phase.

(ii) Scaling. As a result of the optimizations presented in Section 7.4.1, scaling is performed through bit shift, which can be evaluated in GC with no additional communication cost. Scaling converts the bitwidth of the shared variables from b to b_{inp}, where, b_{inp} is the input bitwidth of the next CONV/FC layer. This allows us to significantly reduce the secure execution cost of nonlinear layers in GC.

(iii) MaxPool and ReLU. An MP operation with a window size of $k \times k$ requires $k^2 - 1$ comparison and multiplexing operations, each of which incurs a communication cost of $2 \cdot \kappa \cdot b_{inp}$ bits. Note that MP (ReLU (x))=ReLU (MP (x)), thus we perform MP before ReLU as it shrinks the size of the activation tensor by a factor of k^2, thereby reducing the ReLU cost. Each ReLU operation includes b_{inp} AND operations each of which requires communication of $2 \cdot \kappa \cdot b_{inp}$ bits.

(iv) GC to AS Conversion. For this operation, Alice generates a random b-bit integer which is added to the ReLU output and the sum is revealed to Bob. This operation does not require COT since there is no input from Bob. During conversion, the values are sign-extended to b bits to match the AS ring size. It is worth noting that HE-GC conversion requires 3× more

computation/communication compared to AS-GC conversion. Our computations in the AS domain are performed modulo 2^b and the GC circuit for b-bit addition automatically takes care of the modulo operation. On the contrary, to benefit from SIMD operations in HE, the modulus is chosen as a prime number. Therefore, the circuit for modular addition requires b-bit addition, subtraction, and multiplexing, thereby increasing the cost of HE-GC conversion [188].

Table 7.2. Cost of different phases of linear layers in COINN and previous works. N_{slot} is number of slots in vectorized HE operations. $CostMult(q)$ is cost of one scalar multiplication in $\mathcal{Z}_q$ in HE. q is cipher-text modulus which is $\sim 3\times$ larger than plain-text modulus $p \approx 2^{b_{acc}}$.

Work	Per-layer Complexity	
	One time setup	**Per-inference**
Gazelle/Delphi/CTF2 (HE)	-	$O(\frac{MNL}{N_{slot}}).CostMult(q)$
MiniONN/CTF2 (OT)	-	$O\big(MNb_{acc}(\kappa + Lb_{acc})\big)$
XONN (GC)	-	$O\big(MNLb_{acc}^2\kappa\big)$
COINN – regular (OT)	$O\big(MNb_{acc}\kappa\big)$	$O\big(MNLb_{acc}^2\big)$
COINN – factored (OT)	$O\big(MVb_{acc}\kappa\big)$	$O\big(MVLb_{acc}(N + b_{acc})\big)$

7.5.4 Cost Breakdown and Comparison with Previous Works

To explain the source of run-time improvement in the proposed method, we summarize the cost complexity of different phases of execution of the linear layers in COINN and compare them with previous works in Table 7.2. For HE-based works, the complexity refers only to the computation cost. For OT-based works, the complexity refers to both communication and computation cost, even though the communication is usually the dominant factor. The number of communication rounds for all works is equal to the number of layers, except for XONN which has constant number of rounds.

The performance gains of COINN over prior work stem from two main reasons:

- **Separating setup time.** We move a large part of the computation/communication of the (OT-based) linear layers of COINN to a one-time setup phase without affecting security.

In contrast, previous OT-based methods (MiniONN [187], CrypTFlow2 [212]) repeat these operations for every inference [3]. Moreover, separation of setup and per-inference phases is not readily applicable in the HE-based methods (Gazelle [188], Delphi [1], CrypTFlow2 [212]) or GC-based methods (XONN [3]).

- **Optimizing parameters.** Prior works use a large but fixed bitwidth b_{acc} for all linear layers. COINN customization finds smaller values of b_{acc} that vary from one layer to another, significantly reducing the secure execution cost while preserving the accuracy. Additionally, our factored matrix multiplication can further reduce the execution cost of linear layers when $V < b_{acc}$. Note that by reducing the computational bitwidth, COINN also reduces the cost of protocol conversion and nonlinear layers as formalized in Table 7.1.

7.6 Oblivious BNN Inference

BNNs were originally introduced to minimize memory footprint and computation overhead of plaintext inference. As an added benefit, they also enable fast oblivious inference. While some convenient properties of BNN have been exploited by earlier work on oblivious inference – XONN [3], it still did not exploit the full potential. The primary limitation of XONN was employing a single protocol – GC, which is significantly less efficient than AS for the arithmetic computation in the linear layers. In this work, we improve the performance of oblivious BNN inference by customizing our mixed protocol system presented in the previous section for BNN. In BNN, the weights are forced to be +1 or -1. This property ensures that the vector dot product can be computed through only conditional additions, which in turn is computed through OT in oblivious inference. In the non-linear layers, we show how the Binary Activation (BA) operation in BNNs can be performed free of cost by absorbing its computation in the AS-GC conversion.

[3] The preprocessing phase of Delphi [1] is equivalent to our offline phase and needs to be repeated per inference to ensure security.

Algorithm 3: Protocol for secure binary matrix multiplication

Input : From Alice, weight matrix $[\![W]\!]_A \in \{-1,+1\}^{M\times N}$
Input : From Bob, share of the activation $[\![X]\!]$: $[\![X]\!]_B \in \mathbb{Z}_{2^b}^{N\times L}$
Output : Matrix Product $[\![Z]\!] = [\![W]\!]_A [\![X]\!]_B \in \mathbb{Z}_{2^b}^{M\times L}$
OT message received by Alice, $\left\{\mu'^{(l)} \in \mathbb{Z}_{2^b}^{M\times N}\right\}_{l\in[L]}$
OT messages chosen by Bob, $\left\{\mu_{\mathbf{0}}{}^{(l)} \in \mathbb{Z}_{2^b}^{M\times N}\right\}_{l\in[L]}$, $\left\{\mu_{\mathbf{1}}{}^{(l)} \in \mathbb{Z}_{2^b}^{M\times N}\right\}_{l\in[L]}$

1 Bob randomly generates $[\![\mathbf{R}]\!]_B \in \mathbb{Z}_{2^b}^{M\times N\times L}$
2 **foreach** $m \in [M], n \in [N]$ **do**
3 Alice and Bob run OT_{Lb}^2 where
4 Bob acts as sender with messages $\left\{\mu_{\mathbf{0}}[m,n]^{(l)}, \mu_{\mathbf{1}}[m,n]^{(l)}\right\}_{l\in[L]} = [\![\mathbf{R}]\!]_B[m,n,l] \pm [\![X]\!]_B[n,l]$
5 Alice acts as receiver with choice bits $\frac{W[m,n]+1}{2}$ and receives $\left\{\mu'^{(l)}[m,n]\right\}_{l\in[L]} = \left\{[\![\mathbf{R}]\!]_B[m,n,l] + W[m,n]\cdot[\![X]\!]_B[n,l]\right\}_{l\in[L]}$
6 Alice sets $[\![Z]\!]_A[m,l] = \sum_{n=1}^{N}\left(\mu'^{(l)}[m,n]\right); \forall m \in [M], l \in [L]$
7 Bob sets $[\![Z]\!]_B[m,l] = -\sum_{n=1}^{N}\left([\![\mathbf{R}]\!]_B[m,n,l]\right); \forall m \in [M], l \in [L]$

7.6.1 Binary Matrix Multiplication

Algorithm 3 presents the secure matrix multiplication protocol for the class of binary weights. There are three differences compared to Algorithm 2 for factored matrix multiplication. (i) the unique space of the weight matrix W is simply $\{-1,+1\}$. (ii) we do not need to explicitly maintain a one-hot encoding of W since it can be computed with the simple equation at line 5 of Algorithm 3. (iii) we do not need the two step process of computing partial sums first, and then multiplying them with the unique space later. Instead, Alice and Bob engage in regular OT (as opposed to COT in Algorithm 2), where BOB chooses his messages in accordance with the constant unique space $\{-1,+1\}$ at line 4 of Algorithm 3. Note that both weight and activation are ± 1 in BNNs. However, to ensure the correctness of the accumulation operation during computation of the vector dot product, we need to store the activations $[\![X]\!]$ in $b(=\lceil Log(N)+1\rceil)$ bit variables in Algorithm 3.

7.6.2 Nonlinear Layers

Similar to generic DNNs, upon completion of the linear layers, the result is securely converted from AS to GC for computation of the non-linear layers. In case of BNN, the matrix product Y in $\mathbb{Z}_{2^b}$ also needs to be converted back to the $\{+1,-1\}$ domain. This conversion is performed as the cascade of BN and BA: $sign(\alpha Y+\beta) = sign(Y+\frac{\beta}{\alpha})$, where α and β are BN parameters. Since both α and β is known to Alice (the server), she can locally compute the private variable $[\![\eta]\!]_A = \frac{\beta}{\alpha}$. Then Alice and Bob can perform AS-GC conversion, BN, and BA together by computing $sign([\![Y]\!]+[\![\eta]\!]_A) = sign([\![Y]\!]_B+[\![Y]\!]_A+[\![\eta]\!]_A)$ through GC, where Alice inputs $[\![Y]\!]_A+[\![\eta]\!]_A$ and Bob inputs $[\![Y]\!]_B$. The cost of this computation is exactly the same as the cost of BA performed in the previous work XONN [3] on oblivious BNN inference based entirely on the GC protocol. Therefore, in our mixed protocol solution, we do not pay additional cost for the AS-GC conversion.

7.6.3 Training Adaptive BNN

One major challenge in BNN is ensuring inference accuracy compared too the non-binarized model. In this work, we improve the accuracy of the base BNN by multiplying its width, e.g., by training an architecture with twice as many neurons at each layer. In this realm, we find Slimmable Networks [213], a method of training dynamic DNN [214], quite compatible to our problem setting and adapt them to BNNs. We train a single network with a certain maximum width, say 4× the base network, in a way that the model can still deliver acceptable accuracy at lower widths, e.g., 1× or 2× the base network. After training, we can run oblivious inference with any of the selected widths, thus, providing a tradeoff between accuracy and runtime.

7.7 Evaluation of COINN: Generic DNN Inference

In this section, we empirically evaluate the performance of COINN in various settings. We perform a detailed study of the efficiency gains achieved by each of COINN optimizations,

Table 7.3. COINN benchmarks.

Model	Layers	Acc	MACs	Params
MiniONN [187]	6 CONV, 1 FC, 2 MP, 6 ReLU	88.3	6.1e7	1.6e5
ResNet32	31 CONV, 1 FC, 1 AP, 31 ReLU	68.7	6.9e7	4.7e5
ResNet110	109 CONV, 1 FC, 1 AP, 109 ReLU	94.1	2.5e8	1.7e6
ResNet50	49 CONV, 1 FC, 1, MP, 1 AP, 49 ReLU	76.1	4.1e9	2.5e6

namely, quantization, clustering, and end-to-end parameter configuration, in Section 7.7.1. Next, we provide a side-by-side comparison of COINN with recent works in Section 7.7.2, in terms of the ciphertext execution time, showing 4.7×–36.8× faster inference on contemporary DNNs in LAN setting. We further show that COINN achieves better performance compared to prior work in the high-latency setting.

Evaluation Setup. We use the PyTorch library for training the `FP32` DNNs and develop our security-aware quantization, clustering, and automated parameter configuration with PyTorch backend for easy utilization by the community. Our ciphertext execution uses OT, and CS-PRNG implementations from EMP-toolkit [35] and GC implementation from TinyGarble2 [10]. For fast matrix-multiplication, we utilize the Intel intrinsic instructions and represent matrices with the Eigen library [215].

We run our ciphertext evaluations using 4 threads on machines with 2.2 GHz Intel Xeon CPU and 16 GB RAM. For runtime measurements, we consider two real-world network settings, namely LAN with a throughput of 1.25 GBps, round trip time of 0.25ms, and WAN with a throughput of 125 MBps, round trip time of 100ms. We simulate the network settings via Linux Traffic Control[4].

Benchmarks. We perform evaluations on the CIFAR-10, CIFAR-100, and ImageNet classification benchmarks. The number of classes in these datasets are 10, 100, and 1000, respectively. Table 7.3 presents details of our benchmark DNNs and their `FP32` accuracies. We evaluate the 7-layer network from MiniONN [187] and ResNet110 on CIFAR-10, ResNet32

[4]https://man7.org/linux/man-pages/man8/tc.8.html

on CIFAR-100, and ResNet50 on ImageNet dataset. Our benchmarks cover a wide range of parameter sizes (0.5M to 23M) and number of MAC operations (60M to 4B) commonly observed in real-world models.

Accuracy Measurement. Throughout the evaluations, we report the secure model accuracy, which is measured efficiently (and correctly) by simulating ciphertext operations in PyTorch. The correctness is validated by matching all DNN layers' activations in secure inference with those from PyTorch on randomly selected inputs.

7.7.1 Evaluation of COINN Optimizations

In this section, we provide a breakdown of the savings in secure execution cost as a result of COINN's model adjustment methods and protocol optimization.

Low-Bit Heterogeneous Quantization. We illustrate the benefits of our quantization scheme in reducing the secure communication cost, while maintaining accuracy, for a large scale real-world DNN – ResNet32. Figure 7.5 presents the communication cost and accuracy of secure execution as a function of the bitwidth. The numerical labels on the horizontal axis represent homogeneous quantization (equal bitwidths across all layers), where each label is $b_{inp} = b_w$ with b_{acc} set to $2b_{inp} + 1$. The label 16 represents the configuration implemented in prior works [1, 5] which we use as a baseline. Figure 7.5 shows that while reducing bitwidth in homogeneous setting results in a linear reduction of ciphertext communication, it also results in a significant drop in accuracy.

To mitigate the undesireable accuracy drop of homogeneous quantization, our automated parameter configurator finds a heterogeneous allocation of per-layer bitwidths that simultaneously ensures high accuracy and low communication cost. The rightmost label, Q in Figure 7.5, represents the COINN optimized model with heterogeneous quantization bitwidths across layers. This optimal set of bitwidths results in a communication cost equivalent to the 6-bit homogeneous model and achieves an accuracy comparable to the 16-bit baseline. Such optimization of per-layer bitwidths is made possible via our secure computation-aware quantization which accurately

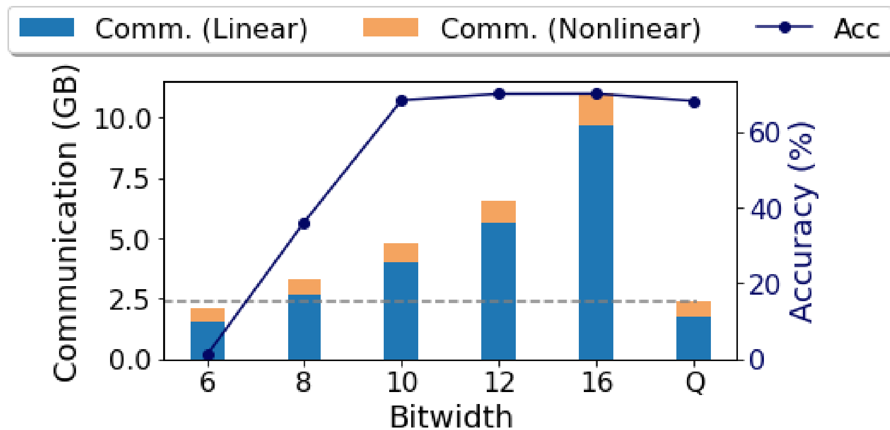


Figure 7.5. Effect of quantization bitwidth on communication cost (bars) and accuracy (curve). The numbers on the horizontal axis show the bitwidth for homogeneous quantization of weights/inputs across all layers. Q represents the heterogeneous bitwidths found by COINN.

simulates the effect of low-bit quantization in ciphertext. This allows us to explore the trade-off between communication cost and model accuracy. We present the heterogeneous bitwidths found by COINN configurator for ResNet32 in Figure 7.6-a.

Factored Matrix-Multiplication. Figure 7.5 shows that the bulk of total communication cost in a quantized model corresponds to linear operations. We now showcase how COINN further reduces this cost via factored matrix-multiplication. Figure 7.7 presents the communication cost and accuracy as a function of the number of unique elements in each layer's weight matrices V. The label Q represents our model with heterogeneous quantization bitwidths from Figure 7.5. The numeric labels to its left represent models with a uniform selection of V across all layers. Such naïve selection results in accuracy degradation, particularly for small V. Our automated parameter configurator finds a heterogeneous allocation of V across DNN layers that balances the tradeoff between inference accuracy and ciphertext communication. The result is an optimal DNN represented with the label Q+C that reduces the secure communication cost of the quantized model by 1.4× while maintaining the original model accuracy. We present the heterogeneous number of per-layer clusters found by our configurator for this benchmark in Figure 7.6-b.

Holistic Optimization. Figure 7.8 presents the reduction in communication cost achieved

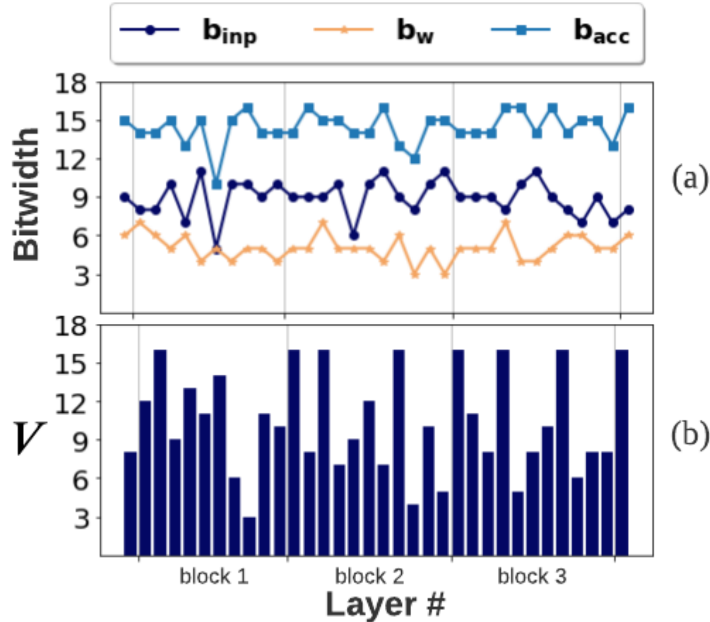


Figure 7.6. Heterogeneous parameters across ResNet-32 layers found by COINN configurator. (a) Quantization bitwidths. (b) Number of clusters V.

by applying COINN automated quantization and clustering on all benchmarks. As our baseline design, we adopt the bitwidths from prior work [1], i.e., 16-bit inputs/weights and 32-bit activations, and perform regular matrix-multiplication. For COINN results, we first find heterogeneous quantization configurations using our genetic algorithm and fine-tune the model to regain accuracy. We show the optimized quantized model via Q on Figure 7.8. Next, we use our automated parameter configurator to find the weight clusters for each layer and fine-tune the resulting model once more to obtain the DNN labeled Q+C. The linear operations in the Q and Q+C settings are performed via regular and factored Matrix-Multiplication, respectively. As seen, by finding the best set of heterogeneous bitwidths across DNN layers, COINN successfully reduces the secure communication for linear and nonlinear layers by 3.9×–4.3× and 1.9×–2.2×, respectively. By optimizing the weight clusters, we further push the efficiency gains on linear layers to 4.8×–8.1×.

Table 7.4 provides the total runtime and communication cost of our baseline, Q, and Q+C configurations in both LAN and WAN settings. The evaluation verifies the effect of our

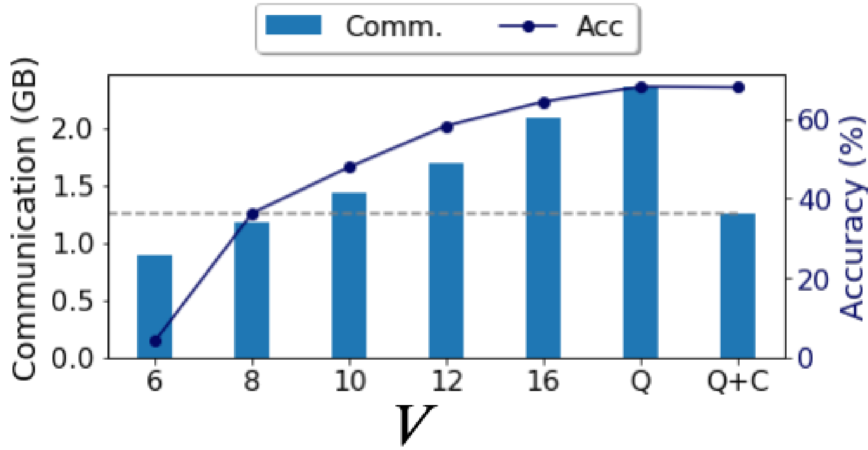


Figure 7.7. Effect of factored multiplication on inference accuracy and communication cost of linear operations. Q represents the baseline quantized DNN. Numbers to its left represent homogeneous V for all layer weights. Q+C represents heterogeneous V configuration found by COINN.

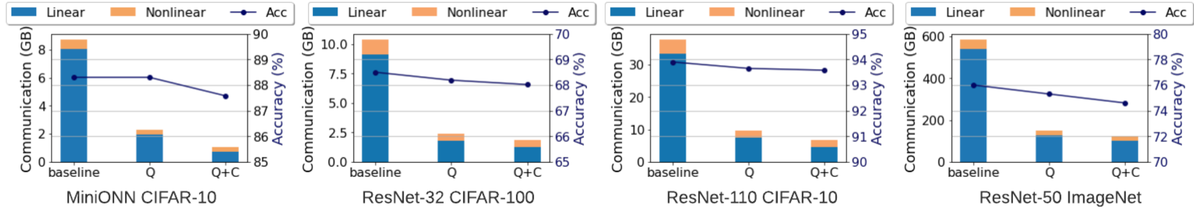


Figure 7.8. Communication for baseline and COINN optimized models, where Q represents quantized model and Q+C further applies clustering to enable factored multiplication.

optimization on the runtime: applying Q+C reduces the baseline runtime by 2.6×–3.9× and 2.3×–4.2× in LAN and WAN settings, respectively. The effect of COINN optimizations on standalone micro-benchmarks of the CONV and ReLU is presented in Appendix 7.7.4.

Setup Time Separation. Finally, we evaluate the effect of introducing the one-time setup phase to reduce the amortized per-inference cost. The setup phase is only performed the first time a connection is established between the client and server and is independent of the number of inferences. In the previous section (Table 7.2), we showed the complexity of linear layers in the setup and per inference phases. We now show the effect of this optimization through experimental evaluation. Figure 7.9 presents the breakdown of setup time and amortized inference time for

Table 7.4. Evaluation of COINN in LAN and WAN settings. Q and C denote quantization and clustering, respectively.

Model	Comm. (GB)			LAN Time (s)			WAN Time (s)		
	Base	Q	Q+C	Base	Q	Q+C	Base	Q	Q+C
MiniONN	8.7	2.3	1.0	4.85	1.9	1.45	74.6	26.5	18.5
Res32	10.4	2.4	1.9	9.8	3.8	3.68	143.9	67.1	62.9
Res110	37.6	9.7	6.8	36.0	14.2	14.0	518.1	242.8	226.0
Res50	583.1	148.0	122.0	571.46	165.3	145.7	4994	1420.4	1189.7

each of the four benchmarks under LAN and WAN settings. As expected, separating the setup time from oblivious inference significantly reduces the runtime.

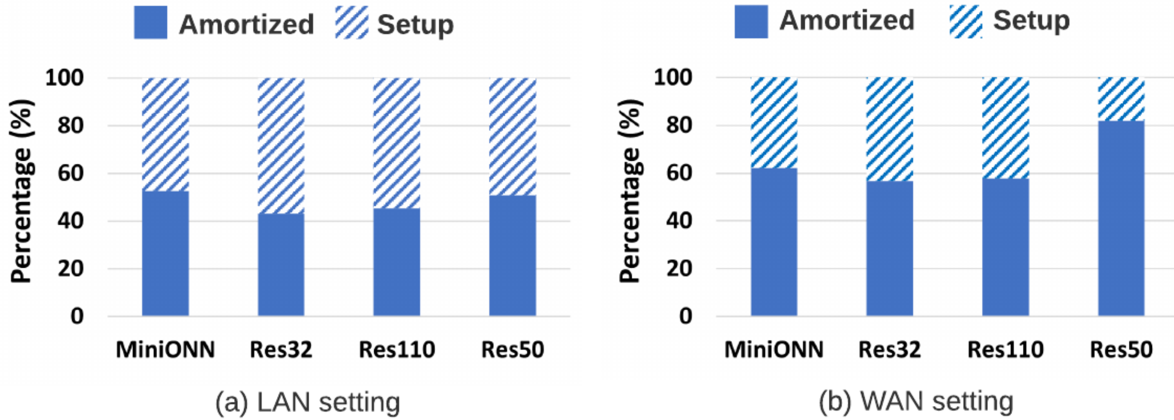


Figure 7.9. Breakdown of setup and amortized times for the under LAN and WAN settings.

7.7.2 Comparison with Prior Work

In this section, we compare COINN amortized runtime with the prior art in oblivious inference. In Table 7.5, we report the performance of COINN along with four contemporary works, namely, XONN [3] with extremely low-bit (binary) weights/activations, Delphi [1] with a hybrid HE-GC protocol, SafeNet [2] which perform ML optimization for Delphi's secure protocol, and CrypTFlow2 [5] which is the current state-of-the-art in oblivious inference. For a fair and accurate comparison, we re-run the open-source codes provided by Delphi[5] and CrypTFlow2[6] to

[5]https://github.com/mc2-project/delphi
[6]https://github.com/mpc-msri/EzPC/tree/master/SCI

Table 7.5. Performance comparison of COINN with best prior work. "Improv." shows the improvement in total runtime. CTF2 refers to CrypTFlow2 [5].

		LAN		WAN		Acc.
		Runtime (s)	Improv.	Runtime (s)	Improv.	(%)
MiniONN	XONN	33.5	23.1×	-	-	83.0
	Delphi	49.9	34.4×	59.8	3.2×	82.9
	SafeNet	53.4	36.8×	-	-	85.1
	CTF2 (HE)	20.8	14.4×	55.4	3.0×	86.0
	CTF2 (OT)	11.9	8.2×	108.2	5.8×	86.0
	COINN	1.45	1×	18.5	1×	87.6
Res32	Delphi	88.8	24.0×	145.9	2.3×	65.7
	SafeNet	128.0	34.6×	-	-	67.5
	CTF2 (HE)	32.6	8.8×	136.9	2.2×	68.0
	CTF2 (OT)	18.7	5.1×	176.7	2.8×	68.0
	COINN	3.7	1×	62.9	1×	68.1
Res110	CTF2 (HE)	110.3	7.8×	448.2	2.0×	94.1
	CTF2 (OT)	65.4	4.7×	579.3	2.6×	94.1
	COINN	14.0	1×	226.0	1×	93.4
Res50	CTF2 (HE)	893.2	6.1×	1463.3	1.2×	76.1
	CTF2 (OT)	1139.8	7.8×	4241.8	3.6×	76.1
	COINN	145.7	1×	1189.7	1×	73.9

obtain runtime/communication measurements on our machines. For the remaining works [3, 2], we directly report the numbers from the original papers since no public code was available.

Table 7.5 shows COINN achieves 4.7×–36.8× faster ciphertext execution in the LAN setting compared to prior work. Even though in the high latency setting the benefit margins are smaller, COINN still outperforms the best methods to date. This is achieved by optimizing both non-linear and linear computations/communications through quantization and factored multiplication. Furthermore, COINN achieves 0.6%– 4.7% higher accuracy with 23.1×–36.8× faster secure runtime compared to prior crypto/ML co-optimization work, namely [3, 1, 2].

Evaluation on Large-scale Benchmarks. To fully demonstrate the efficacy and scalability of COINN model adjustment techniques and custom secure protocols, we evaluate two exceptionally complex DNNs, namely, ResNet110 on CIFAR-10 and ResNet50 on ImageNet datasets. The first benchmark, i.e., ResNet110, is challenging due to the extremely high di-

mensionality of the parameter configuration space: there are 330 bitwidths and 110 clustering parameters that require per-layer adjustment. The second benchmark, i.e., ResNet50, is the largest DNN ever studied in the secure computation domain with over 4 Billion scalar multiplications and additions.

In Table 7.5, we present the runtime for the large scale networks and compare our results with the state-of-the-art CrypTFlow2. In the LAN setting, COINN achieves 4.7×–7.8× and 6.1×–7.8× runtime improvement compared to CrypTFlow2's OT-based and HE-based implementations, respectively. In the WAN setting, COINN achieves 2.6×–3.6× and 1.2×–2× runtime improvement compared to CrypTFlow2's OT-based and HE-based implementations, respectively. It is worth noting that the relatively lower improvement margin achieved by COINN in one specific setting (1.2× for ResNet50, WAN, HE) is due to the heavy imbalance of the cost towards linear layers in this particular benchmark.

7.7.3 Model Customization Runtime

COINN plaintext model customization (Section 7.4) is a one-time process performed on the pre-trained model by the server irrespective of the number of inference or the number of clients. Table 7.6 outlines the runtime of each customization step on one GPU, across various benchmarks. For better comparison, we normalize the customization and fine-tuning runtimes by the time required for training the baseline DNN on the same hardware. For fine-tuning, the number of fine-tuning epochs for each model is determined such that the validation accuracy reaches a convergence plateau. Regarding model customization, we terminate the genetic algorithm when the best obtained score does not improve for more than 5 iterations. Note that COINN customization step enjoys a linear speedup as the number of GPU cores increases. This is due to the independence of score evaluations inside a population [216].

Table 7.6. Runtime of COINN model customization and fine-tuning, normalized by the target DNN's training time on one NVIDIA Titan XP GPU. Here, Q and C denote the quantization and clustering stages, respectively.

Model	**Training** (minutes)	**COINN Steps**	**Customization**		**Fine-tuning**	
			# iter	Runtime	# iter	Runtime
MinioNN	11.6	Q	30	2.29×	20	0.50×
		C	20	2.12×	5	0.14×
Res32	34.2	Q	20	1.48×	20	0.56×
		C	30	1.94×	20	0.65×
Res110	107.6	Q	30	1.89×	5	0.14×
		C	30	1.43×	20	0.59×
Res50	14,040.0	Q	20	0.05×	5	0.14×
		C	30	0.16×	2	0.07×

7.7.4 Evaluation on Microbenchmarks

We present the evaluation results on standalone linear and nonlinear layers of COINN in tables 7.7, 7.8, and 7.9. Observe that the run-time for the convolution layers with regular matrix-multiplication increases quadratically with the bitwidth b. The quantization and automated parameter configurator of COINN thus greatly enhance the performance by minimizing the bitwidths. The run-time of convolution layers with factored multiplication increases linearly with the number of unique elements, Vm and becomes equal to that of regular matrix-multiplication when $V = b$. This is consistent with the analysis in Section 7.5.2. Table 7.9 shows the run-time of combined AS to GC, ReLU, and GC-AS operations. As expected, the run-time increases linearly with b.

Table 7.7. Evaluation on convolution layers of TytaNN with regular matrix multiplication

Input	Kernal	LAN			WAN		
$C \times H \times W$	$N \times F \times F$	$b = 8$	$b = 16$	$b = 32$	$b = 8$	$b = 16$	$b = 32$
$16 \times 32 \times 32$	$16 \times 3 \times 3$	0.021	0.073	0.317	0.703	1.217	3.557
$32 \times 16 \times 16$	$32 \times 3 \times 3$	0.021	0.071	0.284	0.711	1.202	3.350
$64 \times 8 \times 8$	$64 \times 3 \times 3$	0.027	0.070	0.268	0.703	1.220	3.139

Table 7.8. Evaluation on convolution layers of TytaNN with factored matrix multiplication, $b = 16$

Input	Kernal	LAN			WAN		
$C \times H \times W$	$N \times F \times F$	$V = 8$	$V = 12$	$V = 16$	$V = 8$	$V = 12$	$V = 16$
$16 \times 32 \times 32$	$16 \times 3 \times 3$	0.039	0.056	0.073	0.902	1.107	1.217
$32 \times 16 \times 16$	$32 \times 3 \times 3$	0.037	0.054	0.073	0.810	1.011	1.203
$64 \times 8 \times 8$	$64 \times 3 \times 3$	0.038	0.053	0.071	0.812	1.013	1.227

Table 7.9. Evaluation on ReLU of TytaNN (including AS-GC conversions)

Input	LAN			WAN		
$C \times H \times W$	$b = 8$	$b = 16$	$b = 32$	$b = 8$	$b = 16$	$b = 32$
$16 \times 32 \times 32$	0.083	0.182	0.307	1.125	1.569	2.583
$32 \times 16 \times 16$	0.062	0.085	0.16	0.820	1.061	1.565
$64 \times 8 \times 8$	0.026	0.043	0.066	0.708	0.913	1.124

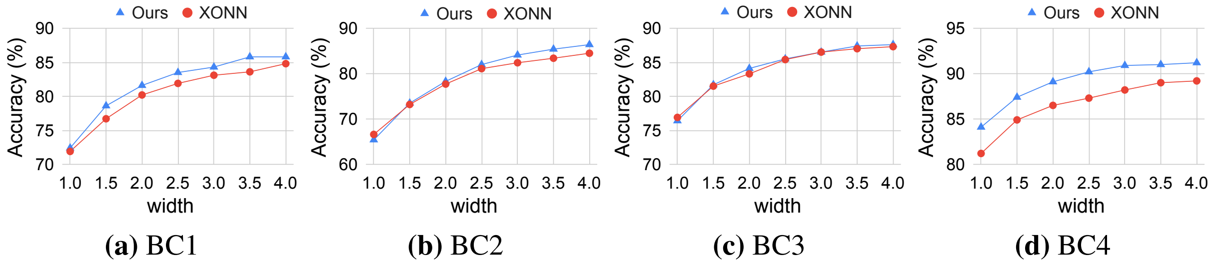
7.8 Evaluations of BNN Inference

Standard Benchmarks. We perform our evaluation on several networks trained on the CIFAR-10 dataset, shown in Table 7.10. These benchmarks provide us with a rich set of comparison baselines as they are commonly used in prior work. Specifically, the BC1 network has been evaluated by the majority oblivious inference papers [187, 217, 218, 188, 3, 1, 5, 2, 4]. Other models are evaluated by XONN [3], the state-of-the-art for oblivious inference of *binary* networks. For brevity, we omit details about layer-wise configurations and refer curious readers to [3] for further information.

Table 7.10. Summary of the trained binary network architectures evaluated on the CIFAR-10 dataset

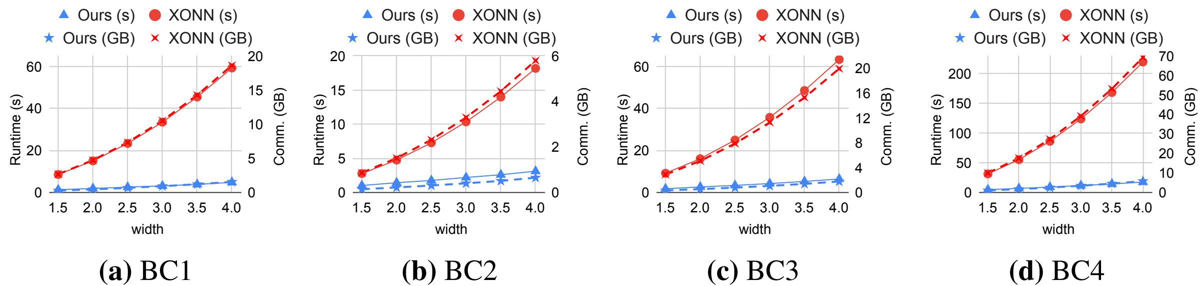
Arch	**Previous Papers**	**Description**
BC1	[187], [217], [218], [188], [3], [1], [5], [2], [4]	7 CONV, 2 MP, 1 FC
BC2	[3]	9 CONV, 3 MP, 1 FC
BC3	[3]	9 CONV, 3 MP, 1 FC
BC4	[3]	11 CONV, 3 MP, 1 FC

Training. For all benchmarks, we use standard backpropagation algorithm proposed



(a) BC1 **(b)** BC2 **(c)** BC3 **(d)** BC4

Figure 7.10. CIFAR-10 test accuracy of each architecture at different widths. Our Adaptive BNN trains a single network that can operate at all widths, whereas previous work (XONN) trains a separate BNN per width



(a) BC1 **(b)** BC2 **(c)** BC3 **(d)** BC4

Figure 7.11. Runtime and communication cost of each architecture at different widths

by [191] to train our binary networks. We split the CIFAR10 dataset to 45k training examples, 5k validation examples, and 10k testing examples, and train each architecture for 300 epochs. We use Adam optimizer with initial learning rate of 0.001, and the learning rate is multiplied by 0.1 after 101, 142, 184 and 220 epochs. The batch size is set to 128 across all CIFAR10 training experiments. The training data is augmented by zero padding the images to 40×40, and randomly cropping a 32×32 window from each zero-padded image.

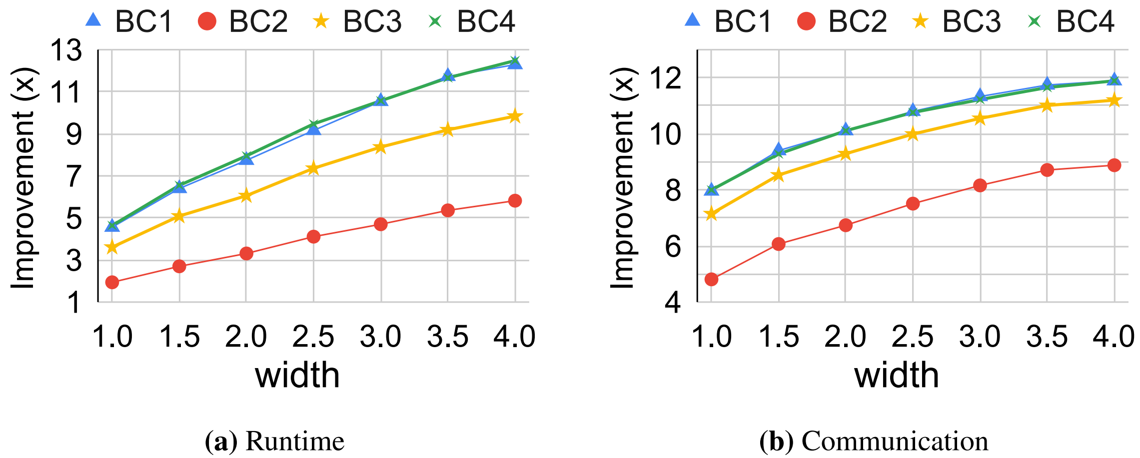
Evaluation Setup. Same as the previous section.

7.8.1 Evaluating Flexible BNNs

Let us start by evaluating our adaptive BNN training. We train slimmable networks with maximum $4\times$ width of the base models presented in Table 7.10. During training, we re-iterate through subsets of widths $\{1\times, 1.5\times, \ldots, 4\times\}$ and perform gradient updates as explained in Section 7.6.3.

Figure 7.10 presents the test accuracy of each network at different widths. We also report the accuracy of independently trained networks reported by XONN. The test accuracy of a particular base BNN architecture can be improved by increasing its width. Our adaptive networks obtain better accuracy than independently trained BNNs at each width. Once the adaptive network is trained, the server can provide oblivious inference service to clients, which we discuss in the following section.

7.8.2 Oblivious Inference



(a) Runtime **(b)** Communication

Figure 7.12. Improvements in LAN runtime and communication compared to XONN. Our protocols achieve 2× to 12× in runtime and 5× to 12× communication reduction

Recall that the runtime of oblivious inference is dominated by data exchange between

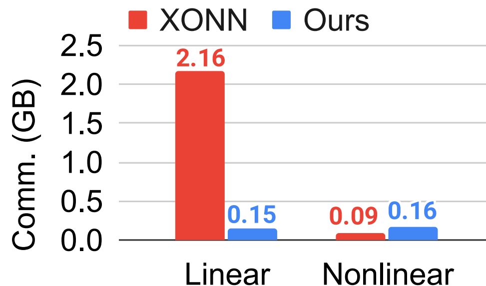


Figure 7.13. Breakdown of communication cost at linear and nonlinear layers for BC2 network. Our protocol significantly reduces XONN's GC-based linear layer cost, with a slight increase in nonlinear layer cost

client and server. We compare the communication cost and runtime of our custom protocol with XONN's GC implementation in Figure 7.11. The horizontal axis in each figure presents the network width. The left and right vertical axes respectively show the runtime (in seconds) and communication (in Giga-Bytes). The figure shows that for all the benchmarks, the runtime and communication of our method are significantly smaller than XONN. As seen, increasing the network width results in higher communication and runtime, which is the cost we pay for higher inference accuracy.

Figure 7.12 summarizes the performance boost achieved by our protocols, i.e., 2× to 12× lower runtime and 5× to 12× lower communication compared to XONN. The enhancement is more significant at higher widths, which shows the scalability for our method. To illustrate the reason behind our protocol's better performance, we focus our attention to the BC2 network at width 2.5, and show the breakdown of its communication cost in Figure 7.13. For the XONN protocol, most of the cost is from linear operations, which we reduce from 2.16GB to 0.15GB. In nonlinear layers, our cost is slightly more that XONN's, i.e., 0.16GB versus 0.09GB, which is due to the extra cost of conversion between AS and GC. Overall, the total communication is reduced from 2.25GB to 0.31GB compared to XONN.

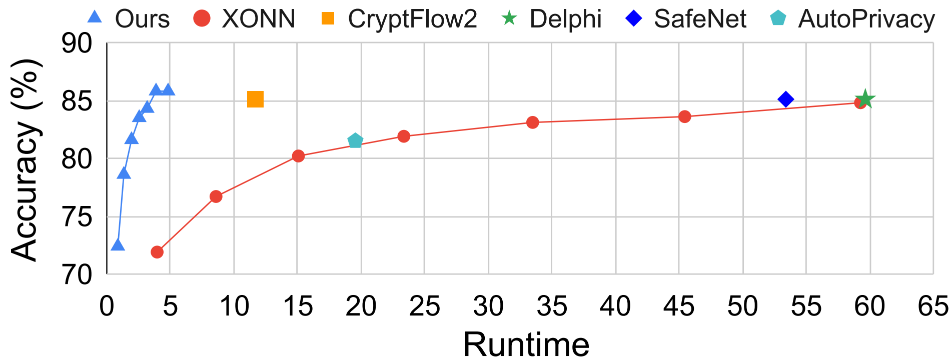


Figure 7.14. Accuracy and runtime of our oblivious BNN inference, compared with contemporary research with same server-client scenario (two-party HbC). XONN [3] evaluates BNNs, whereas Cryptflow2 [5], Delphi [1], SafeNet [2], and AutoPrivacy [4] evaluate non-binary models.

Comparison to Non-binary Models. Among the architectures presented in Table 7.10,

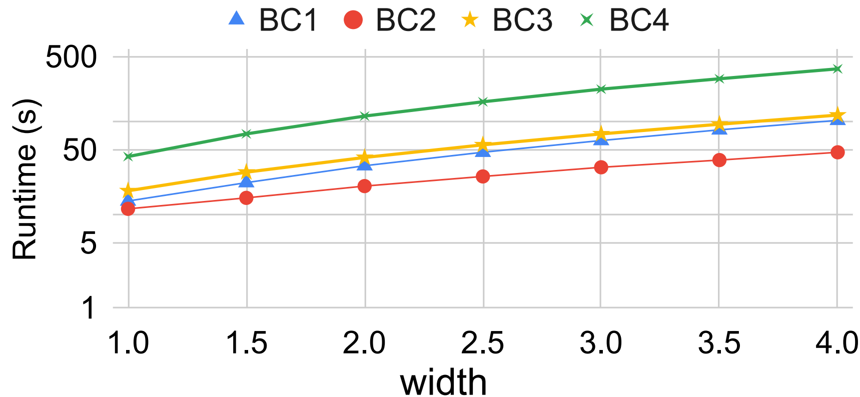


Figure 7.15. Runtime in WAN setting with ∼ 20 MBps bandwidth and ∼ 50 ms network delay

BC1 has been commonly evaluated in contemporary oblivious inference research. In Figure 7.14 we compare the performance of our method to the best-performing earlier work on this benchmark. The vertical and horizontal axes in the figure represent test accuracy and runtime, hence, points to the top-left corner are more desirable. Our method achieves a better accuracy/runtime tradeoff than all contemporary work while providing flexibility. Compared to Cryptflow2 (the most recent oblivious inference framework at the time of this paper), our method achieves 3× faster inference at the same accuracy.

Evaluation in Wide Area Network (WAN). So far we reported our runtimes for the setting where client and server are connected via LAN, which is the most common assumption among prior work. We now extend our evaluation to the WAN setting, where the bandwidth is ∼ 20MBps and the delay is ∼ 50ms. The aforesaid bandwidth and delay correspond to the connection speed between two AWS instances located in "US-West-LA-1a" and "US-East-2a". Runtimes are reported in Figure 7.15, showing varying inference time from 13 to 367 seconds depending on architecture and width. The results show the great potential of BNNs for commercial use. Indeed, the delay introduced by oblivious inference might not be tolerable in many applications that require real-time response, e.g., Amazon Alexa. However, there exist many applications where guaranteeing privacy is much more crucial than runtime, and several seconds or even minutes of delay can be tolerated. We evaluate two such applications in the

following section.

7.9 Summary

In this chapter, we presented oblivious DNN and BNN inference frameworks that outperform state-of-the-art both in accuracy and efficiency. Through a unique combination of complimentary optimizations in ML and crypto domains, this effort brings us one step closer to real life deployment of AI in the privacy-preserving setting. The enhanced performance of this work roots in three innovations, namely, ciphertext-aware quantization, enhanced data reuse, and automated parameter configuration. Our contributions in the plaintext are accompanied by efficient custom cryptographic protocols. We performed rigorous empirical analysis on every step of our optimization process to demonstrate their effect on reducing the secure communication and oblivious inference runtime. Our evaluations on practical DNN benchmarks showed an end-to-end runtime speedup of 3×– 7× over the best prior work. Furthermore, our evaluations on practical BNN benchmarks showed an end-to-end runtime speedup of 2× to 12× over the state-of-the-art in oblivious BNN inference.

Acknowledgement. This chapter, in part, has been accepted to (i) ACM Conference on Computer and Communications Security as: Siam U Hussain, Mojan Javaheripi, Mohammad Samragh, and Farinaz Koushanfar. "COINN: Crypto/ML Codesign for Oblivious Inference via Neural Networks", and (ii) 2021 Conference on Computer Vision and Pattern Recognition (CVPR) as: Mohammad Samragh, Siam U Hussain, Xinqiao Zhang, and Farinaz Koushanfar, "On the Application of Binary Neural Networks in Oblivious Inference". The dissertation author was the primary investigator of both papers.

Chapter 8

Conclusion and Open Challenges

This dissertation takes us one step closer to practical adoption of provable privacy primitives through algorithmic enhancement, abstraction, acceleration through specialized hardware platforms, and application-specific custom protocols. The highlights of this dissertation include the first and currently the only framework supporting scalable execution of Yao's GC protocol in both honest-but-curious and malicious security models, the fastest hardware accelerator for the GC protocol, and the fastest oblivious DNN inference engine. We conclude this dissertation by outlining a couple of possible future directions.

Automated Mixed Protocol Compilation. Over the last two decades, diligent efforts by the researchers have resulted in a diverse set of privacy-preserving primitives. These primitives differ in capabilities, performance, and applicability in different scenarios. There is no single primitive that provide the best performance in all possible scenarios and operations. This dissertation contributes to the effort of improving scalability and efficiency of Multi-Party Computation (MPC) protocols, especially GC and Arithmetic Sharing (AS) protocols. In parallel to these, the field of Homomorphic Encryption (HE) has seen tremendous growth in speed and capabilities. At present, researchers overwhelmingly adopt a mixed protocol system where the most suitable primitives are employed for different parts of the computation. However, in these works, the combinations of these protocols are chosen manually, which requires a thorough understanding of all the available primitives. Even then, there remains the possibility that there

exists a better combination. An exciting research problem in this domain is to devise an algorithm that will accept a function as input and automatically compile it to an optimized mixed protocol system. An added complexity to this problem is that the optimal combination of the primitives also depends on the computation platform and communication channel held by the relevant parties besides the given function. Finally, since privacy-preserving computation is a fast-evolving field, one crucial property of the algorithm is flexibility and modularity to allow adaptation to the advancements in the field.

Privacy-Preserving Machine Learning (ML) Training. While our work presented in Chapter 7 provides the fastest oblivious ML inference, efficiently training large models while maintaining privacy of the training data still remains a challenge. Current efforts of running the entire training through MPC protocols require several hundred hours to train small to moderate models [190]. Methods based on Federated Learning (FL) have been proposed where the training data is protected through cryptographic methods while the trained weight parameters are protected through Differential Privacy (DP) [219, 220]. The FL methods have presented the best combination of accuracy and efficiency till now, albeit on small-scale models. Moving forward the challenge is to improve upon the scalability of these methods to be able to train and update large models with acceptable speed.

Hybrid Accelerator for Privacy-Preserving Computation. In this dissertation, we presented the fastest hardware accelerator for the GC protocol. In parallel, there has been a substantial research effort in accelerating HE through custom hardware platforms. One missing piece in this domain is perhaps specialized hardware for Oblivious Transfer (OT) which forms the basis of a large variety of MPC protocols. An exciting candidate for hardware acceleration is the Silent OT [221, 222] protocol introduced recently. The primary bottleneck of Silent OT is computation as opposed to communication in the previous variants of OT extension. Furthermore, to enable fast execution of mixed protocol systems, we need to combine these individual accelerators into a unified hybrid one. There are two major motivations of a hybrid

accelerator. First, in a mixed protocol system with accelerators located in different hardware, switching between protocols will incur additional overhead which can be avoided with a unified accelerator. Second, combining different security primitives opens the possibility of shared resources thereby optimizing the overall resource usage. Besides obvious candidates for resource sharing like BRAMs or DSPs, there are possibly certain computations that are common among the different primitives.

Appendix A

Command Line Options and Available Functions in TinyGarble2

Listing 2 shows the command line options to access the TinyGarble2 GC back-end and Listing 3 shows the available functions in the TinyGarble2 program interface.

Listing 2. Command line options to access TinyGarble2 GC back-end

```
./SYGC
-h [--help]                         produce help message
-k [--party] arg (=1)               party id: 1: Alice, 2: Bob
-n [--netlist] arg (=sum_8bit.bin)  netlist file address
-p [--port] arg (=1234)             socket port
-s [--server_ip] arg (=127.0.0.1)   server's IP
-i [--input] arg (=0)               hexadecimal input (little endian)
-j [--init] arg (=0)                hexadecimal init (little endian)
-c [--cycles] arg (=1)              number of cycles to run
-r [--repeat] arg (=1)              number of times to repeat the run
-m [--output_mode] arg (=0)         0: reveal output at every cycle,
                                    1: reveal output at last cycle
                                    2: transfer output at every cycle,
                                    3: transfer output last cycle
-f [--file] arg                     netlist, input, init, cycles, repeat,
                                    output_mode are read from this file
--sh                                honest-but-curious model
```

Listing 3. Available functions in the TinyGarble2 program interface

```
/* '_x' suffix indicates secret variable*/
gc_int(uint8_t owner, uint8_t bit_width, int64_t a)
/*create secret variable. owner = ALICE/BOB/NONE.*/
```

```
gc_int_array(uint8_t owner, uint8_t bit_width, auto A, uint8_t len0, ...)
/*create secret array (up to 4D). owner = ALICE/BOB/NONE.*/
reveal(gc_int a_x, uint8_t bit_width, bool is_signed = true)
/*reveal the secret value of an integer*/
reveal_array(auto& A, auto A_x, uint8_t bit_width, , ...)
/*reveal the secret value of an integer array (up to 4D).*/
sign_extend(gc_int& y_x, gc_int a_x, uint8_t bit_width_target,
                                            uint8_t bit_width)
/*sign extend a_x and store in y_x*/
assign(gc_int& y_x, int64_t a, uint8_t bit_width)
/*y_x = a, a is known to both parties*/
assign(gc_int& y_x, gc_int a_x, uint8_t bit_width)
/*y_x = a_x*/
assign_array(auto& Y_x, auto A, uint8_t bit_width, uint8_t len0,...)
/*assign array (upto 4D), A can be both secret or public.*/
add(gc_int& y_x, gc_int a_x, gc_int b_x, uint8_t bit_width_a,
                                            uint8_t bit_width_b)
/*y_x = a_x + b_x*/
sub(gc_int& y_x, gc_int a_x, gc_int b_x, uint8_t bit_width_a,
                                            uint8_t bit_width_b)
/*y_x = a_x - b_x*/
neg(gc_int& y_x, gc_int a_x, uint8_t bit_width)
/*y_x = -a_x*/
mult(gc_int& y_x, gc_int a_x, gc_int b_x, uint8_t bit_width_y,
                     uint8_t bit_width_a, uint8_t bit_width_b)
/*y_x = a_x * b_x*/
mat_mult(uint8_t row_A, uint8_t inner, uint8_t col_B, auto &C_x, auto &A_x,
            auto &B_x, uint8_t bit_width_A, uint8_t bit_width_B,
            uint8_t bit_width_C, uint8_t bit_width_MAC)
/*C_x = A_x * B_x*/
lt(gc_int& y_x, gc_int a_x, gc_int b_x, uint8_t bit_width_a,
                                        uint8_t bit_width_b)
```

```
/*y_x = a_x < b_x*/
ifelse(gc_int& y_x, gc_int c_x, gc_int a_x, gc_int b_x,
                uint8_t bit_width_a, uint8_t bit_width_b)
/*y_x = c_x? a_x : b_x*/
max(gc_int& y_x, gc_int a_x, gc_int b_x, uint8_t bit_width_a,
                                        uint8_t bit_width_b)
/*y_x = max(a_x, b_x)*/
min(gc_int& y_x, gc_int a_x, gc_int b_x, uint8_t bit_width_a,
                                        uint8_t bit_width_b)
/*y_x = min(a_x, b_x)*/
left_shift(gc_int& a_x, uint8_t shift, uint8_t bit_width)
/*a_x << shift*/
right_shift(gc_int& a_x, uint8_t shift, uint8_t bit_width)
/*a_x >> shift*/
```

Appendix B

Architecture of FASE

Figure B.1 shows an enlarged architecture of FASE (previously shown in Figure 4.2 at Section 4.3).

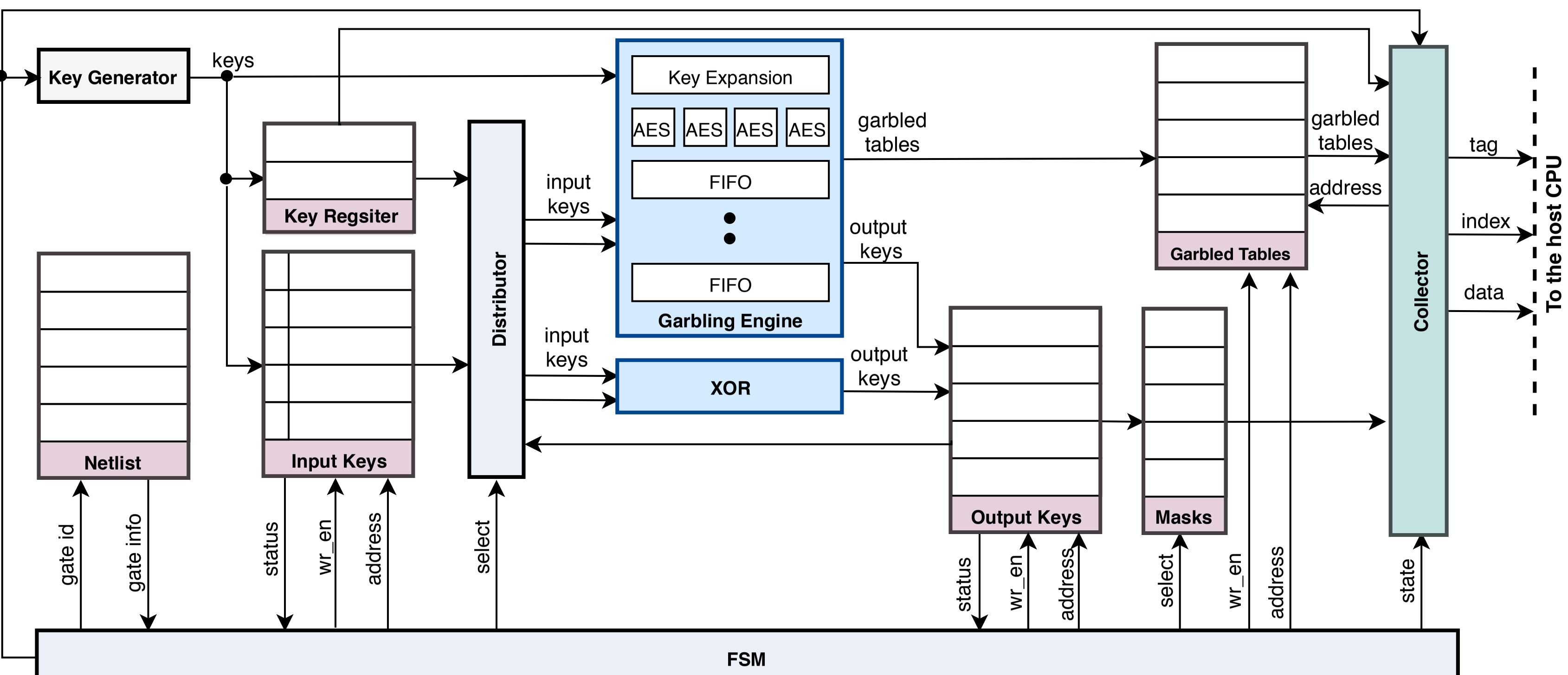


Figure B.1. Enlarged Architecture of FASE

Bibliography

[1] 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 20)*, 2020.

[2] Q. Lou, Y. Shen, H. Jin, and L. Jiang, "SAFENet: A Secure, Accurate and Fast Neural Network Inference," in *International Conference on Learning Representations*, 2021.

[3] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. E. Lauter, and F. Koushanfar, "XONN: XNOR-based Oblivious Deep Neural Network Inference." in *USENIX Security*, 2019.

[4] Q. Lou, B. Song, and L. Jiang, "AutoPrivacy: Automated Layer-wise Parameter Selection for Secure Neural Network Inference," in *Advances in Neural Information Processing Systems*, 2020.

[5] 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*, 2020, pp. 325–342.

[6] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "TinyGarble: Highly compressed and scalable sequential garbled circuits," in *IEEE S&P*, 2015.

[7] A. Yao, "How to generate and exchange secrets," in *Foundations of Computer Science, 1986., 27th Annual Symposium on*, 1986.

[8] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *Symposium on Theory of computing*. ACM, 1990.

[9] S. M. Riazi, M. Javaheripi, S. U. Hussain, and F. Koushanfar, "MPCircuits: Optimized Circuit Generation for Secure Multi-Party Computation," in *Hardware Oriented Security and Trust (HOST)*. IEEE, 2019.

[10] **Siam U Hussain**, B. Li, F. Koushanfar, and R. Cammarota, "TinyGarble2: Smart, Efficient, and Scalable Yao's Garble Circuit," in *ACM Workshop on Privacy-Preserving Machine Learning in Practice(PPMLP)*, 2020.

[11] S. U. Hussain and F. Koushanfar, "FASE: FPGA Acceleration of Secure Function Evaluation," in *Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019.

[12] S. U. Hussain, B. D. Rouhani, M. Ghasemzadeh, and F. Koushanfar, "MAXelerator: FPGA accelerator for privacy preserving multiply-accumulate (MAC) on cloud servers," in *Design Automation Conference (DAC)*. IEEE/ACM, 2018.

[13] M. Naor and B. Pinkas, "Computationally secure oblivious transfer," *Journal of Cryptology*, vol. 18, no. 1, pp. 1–35, 2005.

[14] M. Atallah, M. Bykova, J. Li, K. Frikken, and M. Topkara, "Private collaborative forecasting and benchmarking," in *Proceedings of the 2004 ACM workshop on Privacy in the electronic society*, 2004, pp. 103–114.

[15] X. Fang, S. Ioannidis, and M. Leeser, "Secure Function Evaluation Using an FPGA Overlay Architecture." in *FPGA*, 2017.

[16] S. U. Hussain and F. Koushanfar, "Privacy preserving localization for smart automotive systems," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[17] ——, "P3: Privacy preserving positioning for smart automotive systems," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 23, no. 6, pp. 1–19, 2018.

[18] S. U. Hussain, M. S. Riazi, and F. Koushanfar, "SHAIP: Secure Hamming Distance for Authentication of Intrinsic PUFs," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 23, no. 6, pp. 1–20, 2018.

[19] M. Rostami, F. Koushanfar, and R. Karri, "A Primer on Hardware Security: Threat Models, Metrics, and Remedies," *Proceedings of the IEEE*, 2014, to appear.

[20] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, and F. Koushanfar, "Compacting Privacy-Preserving k-Nearest Neighbor Search using Logic Synthesis," in *DAC*. IEEE, 2015.

[21] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *S&P*. IEEE, 2013.

[22] X. Wang, S. Ranellucci, and J. Katz, "Authenticated garbling and efficient maliciously secure two-party computation," in *CCS*. ACM, 2017.

[23] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending Oblivious Transfers Efficiently." in *Crypto*, vol. 2729. Springer, 2003.

[24] D. Beaver, "Precomputing oblivious transfer," in *Annual International Cryptology Conference*. Springer, 1995, pp. 97–109.

[25] 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 & communications security*, 2013, pp. 535–548.

[26] V. Kolesnikov and T. Schneider, “Improved garbled circuit: Free XOR gates and applications,” in *International Colloquium on Automata, Languages, and Programming*, Springer. Springer, 2008.

[27] M. Naor, B. Pinkas, and R. Sumner, “Privacy preserving auctions and mechanism design,” in *Conference on Electronic Commerce*, 1999.

[28] S. Zahur, M. Rosulek, and D. Evans, “Two halves make a whole,” in *Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 220–250.

[29] B. Kreuter, A. Shelat, B. Mood, and K. Butler, “PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation.” in *USENIX Security*, 2013.

[30] J. Katz, S. Ranellucci, M. Rosulek, and X. Wang, “Optimizing authenticated garbling for faster secure two-party computation,” in *Annual International Cryptology Conference*. Springer, 2018, pp. 365–391.

[31] A. Ben-Efraim, Y. Lindell, and E. Omri, “Optimizing semi-honest secure multiparty computation for the Internet,” in *CCS*. ACM, 2016.

[32] D. Demmler, T. Schneider, and M. Zohner, “ABY-A Framework for Efficient Mixed-Protocol Secure Two-Party Computation.” in *NDSS*. The Internet Society, 2015.

[33] B. Mood, D. Gupta, H. Carter, K. Butler, and P. Traynor, “Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation,” in *EuroS&P)*. IEEE, 2016, pp. 112–127.

[34] N. Büscher, M. Franz, A. Holzer, H. Veith, and S. Katzenbeisser, “On compiling Boolean circuits optimized for secure multi-party computation,” *Formal Methods in System Design*, vol. 51, no. 2, pp. 308–331, 2017.

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

[36] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, 1998.

[37] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” *Proceedings of the IEEE*, vol. 105, no. 12, 2017.

[38] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

[39] S. Zahur and D. Evans, “Obliv-C: A Language for Extensible Data-Oblivious Computation.” *IACR Cryptology ePrint Archive*, vol. 2015, p. 1153, 2015.

[40] J. Bringer, H. Chabanne, and A. Patey, “Shade: Secure hamming distance computation from oblivious transfer,” in *International Conference on Financial Cryptography and Data Security*. Springer, 2013.

[41] M. S. Kiraz, Z. A. Genç, and S. Kardas, “Security and efficiency analysis of the Hamming distance computation protocol based on oblivious transfer,” *Security and Communication Networks*, vol. 8, no. 18, 2015.

[42] Y. LeCun, C. Cortes, and C. Burges, “MNIST handwritten digit database,” *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, vol. 2, p. 18, 2010.

[43] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, “Fairplay – A Secure Two-Party Computation System,” in *USENIX Security*. ACM, 2004.

[44] A. Rastogi, M. A. Hammer, and M. Hicks, “WYSTERIA: A programming language for generic, mixed-mode multiparty computations,” in *S&P*. IEEE, 2014.

[45] J. Boyar and R. Peralta, “Concrete Multiplicative Complexity of Symmetric Functions,” in *MFCS*. Springer, 2006, pp. 179–189.

[46] B. I. C. R. Group, “Semi-Honest-BMR,” https://github.com/cryptobiu/Semi-Honest-BMR, 2016.

[47] D. Demmler, G. Dessouky, F. Koushanfar, A.-R. 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*. ACM, 2015, pp. 1504–1517.

[48] G. Dessouky, F. Koushanfar, A.-R. Sadeghi, T. Schneider, S. Zeitouni, and M. Zohner, “Pushing the communication barrier in secure computation using lookup tables,” in *NDSS*, 2017.

[49] P. Plonski, “keras2cpp,” https://github.com/pplonski/keras2cpp, 2020.

[50] N. Mariella, “From Keras to C,” https://github.com/aljabr0/from-keras-to-c, 2019.

[51] P. Bogetoft, I. Damgård, T. P. Jakobsen, K. Nielsen, J. Pagter, and T. Toft, “A practical implementation of secure auctions based on multiparty integer computation,” in *Financial Cryptography*. Springer, 2006.

[52] Z. Huang, “Privacy Preserving Auction.” 2016.

[53] M. Larson, C. Hu, R. Li, W. Li, and X. Cheng, “Secure auctions without an auctioneer via verifiable secret sharing,” in *Proceedings of the 2015 Workshop on Privacy-Aware Mobile Computing*. ACM, 2015, pp. 1–6.

[54] M. R. Clarkson, S. Chong, and A. C. Myers, “Civitas: Toward a secure voting system,” in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 354–368.

[55] A. Fujioka, T. Okamoto, and K. Ohta, “A practical secret voting scheme for large scale elections,” in *International Workshop on the Theory and Application of Cryptographic Techniques*. Springer, 1992, pp. 244–251.

[56] M. Franz, A. Holzer, S. Katzenbeisser, C. Schallhart, and H. Veith, “CBMC-GC: An ANSI-C Compiler for Secure Two-Party Computations,” in *Compiler Construction*. Springer, 2014.

[57] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, “ObliVM: A programming framework for secure computation,” in *S&P*. IEEE, 2015.

[58] S. U. Hussain, B. D. Rouhani, M. Ghasemzadeh, and F. Koushanfar, “MAXelerator: FPGA Accelerator for Privacy Preserving Multiply-Accumulate (MAC) on Cloud Servers,” in *DAC*. ACM, 2018.

[59] E. M. Songhori, S. Zeitouni, G. Dessouky, T. Schneider, A.-R. Sadeghi, and F. Koushanfar, “GarbledCPU: a MIPS processor for secure computation in hardware,” in *DAC*. ACM, 2016.

[60] V. Nikolaenko, S. Ioannidis, U. Weinsberg, M. Joye, N. Taft, and D. Boneh, “Privacy-preserving matrix factorization,” in *Conference on Computer & communications security*. ACM, 2013.

[61] S. Manuel, “Classification and generation of disturbance vectors for collision attacks against SHA-1,” *Designs, Codes and Cryptography*, vol. 59, no. 1-3, pp. 247–263, 2011.

[62] X. Wang, Y. L. Yin, and H. Yu, “Finding collisions in the full SHA-1,” in *Annual international cryptology conference*. Springer, 2005, pp. 17–36.

[63] A. Satoh, “Hardware architecture and cost estimates for breaking SHA-1,” in *International Conference on Information Security*. Springer, 2005, pp. 259–273.

[64] N. F. Pub, “197: Advanced encryption standard (AES),” *Federal information processing standards publication*, vol. 197, no. 441, 2001.

[65] S. Pu, P. Duan, and J.-C. Liu, “Fastplay-A Parallelization Model and Implementation of SMC on CUDA based GPU Cluster Architecture.” *IACR Cryptology ePrint Archive*, 2011.

[66] N. Husted, S. Myers, A. Shelat, and P. Grubbs, “GPU and CPU parallelization of honest-but-curious secure two-party computation,” in *Computer Security Applications Conference*. ACM, 2013.

[67] K. Wold and C. H. Tan, “Analysis and enhancement of random number generator in FPGA based on oscillator rings,” *International Journal of Reconfigurable Computing*, vol. 2009, 2009.

[68] M. L. Pinedo, *Scheduling: theory, algorithms, and systems*. Springer, 2016.

[69] K. Lakshmanan, S. Kato, and R. R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *2010 31st IEEE Real-Time Systems Symposium*. IEEE, 2010, pp. 259–268.

[70] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 2011, vol. 24.

[71] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distributed Computing*, vol. 59, no. 3, pp. 381–422, 1999.

[72] W. Bożejko, *A new class of parallel scheduling algorithms*. Oficyna wydawn. Politechniki Wrosłwskiej, 2010.

[73] J. R. Goodman and W.-C. Hsu, "Code scheduling and register allocation in large basic blocks," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, 2014, pp. 88–98.

[74] A. Benoit, Ü. V. Çatalyürek, Y. Robert, and E. Saule, "A survey of pipelined workflow scheduling: Models and algorithms," *ACM Computing Surveys (CSUR)*, vol. 45, no. 4, p. 50, 2013.

[75] J. P. Shen and M. H. Lipasti, *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[76] N. Jones *et al.*, "The learning machines," *Nature*, vol. 505, no. 7482, 2014.

[77] J. Kirk, "IBM join forces to build a brain-like computer," http://www.pcworld.com/article/2051501/universities-join-ibm-in-cognitive-computing-researchproject.html, 2016.

[78] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar, "Deep3: Leveraging three levels of parallelism for efficient deep learning," in *DAC*. ACM, 2017.

[79] X. Wang, S. D. Gordon, A. McIntosh, and J. Katz, "Secure computation of MIPS machine code," in *ESORICS*. Springer, 2016.

[80] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft, "Privacy-preserving ridge regression on hundreds of millions of records," in *Symposium on S & P*. IEEE, 2013.

[81] M. Journée, Y. Nesterov, P. Richtárik, and R. Sepulchre, "Generalized power method for sparse principal component analysis," *Journal of Machine Learning Research*, vol. 11, no. Feb, 2010.

[82] C. Fowlkes, S. Belongie, F. Chung, and J. Malik, "Spectral grouping using the Nystrom method," *Trans. on pattern analysis and machine intelligence*, 2004.

[83] L. Deng and D. Yu, "Deep learning: methods and applications," *Foundations and Trends in Signal Processing*, vol. 7, no. 3–4, 2014.

[84] G. Connor, L. R. Goldberg, and R. A. Korajczyk, *Portfolio risk analysis*. Princeton University Press, 2010.

[85] H. Krcmar, R. Reussner, and B. Rumpe, *Trusted cloud computing*. Springer, 2014.

[86] XILLYBUS, "http://xillybus.com/," 2017.

[87] J. Daemen and V. Rijmen, "The Rijndael Block Cipher," 2013.

[88] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications," NIST, Tech. Rep. 800-22, 2001.

[89] Y. LeCun, C. Cortes, and C. Burges, "MNIST dataset," http://yann.lecun.com/exdb/mnist/, 2017.

[90] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy," in *International Conference on Machine Learning*, 2016.

[91] "UCI Machine Learning Repository: ISOLET Data Set," https://archive.ics.uci.edu/ml/datasets/isolet, 2017.

[92] "UCI Machine Learning Repository: Daily and Sports Activities Data Set," https://archive.ics.uci.edu/ml/datasets/Daily+and+Sports+Activities, 2017.

[93] F. Kerschbaum, T. Schneider, and A. Schröpfer, "Automatic protocol selection in secure two-party computations," in *International Conference on Applied Cryptography and Network Security*. Springer, 2014.

[94] X. S. Wang, Y. Huang, T. H. Chan, A. Shelat, and E. Shi, "SCORAM: oblivious RAM for secure computation," in *CCS*. ACM, 2014.

[95] J. A. Varela and N. Wehn, "Near Real-Time Risk Simulation of Complex Portfolios on Heterogeneous Computing Systems with OpenCL," in *International Workshop on OpenCL*. ACM, 2017.

[96] J. Hubaux, S. Capkun, and J. Luo, "The security and privacy of smart vehicles," in *IEEE S & P*, 2004.

[97] "Automotive Security Best Practices - Intel," 2015.

[98] P. Papadimitratos, L. Buttyan, T. Holczer, E. Schoch, J. Freudiger, M. Raya, Z. Ma, F. Kargl, A. Kung, and J. Hubaux, "Secure vehicular communication systems: design and architecture," in *IEEE CM*, 2008.

[99] R. Cheng, Y. Zhang, E. Bertino, and S. Prabhakar, "Preserving user location privacy in mobile data management infrastructures," in *Privacy Enhancing Technologies*. Springer, 2006.

[100] P. Kalnis, G. Ghinita, K. Mouratidis, and D. Papadias, "Preventing location-based identity inference in anonymous spatial queries," in *IEEE ITKDE*, 2007.

[101] M. Gruteser and D. Grunwald, "Anonymous usage of location-based services through spatial and temporal cloaking," in *ICMSAS*. ACM, 2003.

[102] A. Khoshgozaran and C. Shahabi, "Blind evaluation of nearest neighbor queries using space transformation to preserve location privacy," in *ASTD*. Springer, 2007.

[103] G. Zhong, I. Goldberg, and U. Hengartner, "Louis, lester and pierre: Three protocols for location privacy," in *Privacy Enhancing Technologies*. Springer, 2007.

[104] G. Ghinita, P. Kalnis, A. Khoshgozaran, C. Shahabi, and K. Tan, "Private queries in location based services: anonymizers are not necessary," in *SIGMOD ICMD*. ACM, 2008.

[105] M. Atallah and W. Du, "Secure multi-party computational geometry," in *Algorithms and Data Structures*. Springer, 2001.

[106] Y. Huang and R. Vishwanathan, "Privacy preserving group nearest neighbour queries in location-based services using cryptographic techniques," in *IEEE GLOBECOM*, 2010.

[107] Y. Shang, Z. Liu, J. Wang, and X. Xiao, "Triangle and centroid localization algorithm based on distance compensation," in *ICISCE*. IET, 2012.

[108] J. Blumenthal, R. Grossmann, F. Golatowski, and D. Timmermann, "Weighted centroid localization in zigbee-based sensor networks," in *IEEE WISP*, 2007.

[109] J. Zhao, Q. Zhao, Z. Li, and Y. Liu, "An improved Weighted Centroid Localization algorithm based on difference of estimated distances for Wireless Sensor Networks," in *Telecommunication Systems*. Springer, 2013.

[110] J. Zheng, C. Wu, H. Chu, and P. Ji, "Localization algorithm based on RSSI and distance geometry constrain for wireless sensor network," in *IEEE ICECE*, 2010.

[111] P. Bahl and V. Padmanabhan, "RADAR: An in-building RF-based user location and tracking system," in *IEEE INFOCOM*, 2000.

[112] C. Clifton, M. Kantarcioglu, J. Vaidya, X. Lin, and M. Zhu, "Tools for privacy preserving distributed data mining," in *SIGKDD Explorations Newsletter*, 2002.

[113] A. Ranganathan, N. O. Tippenhauer, B. Škorić, D. Singelée, and S. Čapkun, "Design and implementation of a terrorist fraud resilient distance bounding system," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 415–432.

[114] K. B. Rasmussen and S. Capkun, "Realization of RF Distance Bounding." in *USENIX Security Symposium*, 2010, pp. 389–402.

[115] U. D. of Defense, “Global positioning system standard positioning service performance standard,” https://www.gps.gov/technical/ps/2008-SPS-performance-standard.pdf, 2008.

[116] L. Girod, V. Bychkovskiy, J. Elson, and D. Estrin, “Locating tiny sensors in time and space: A case study,” in *DAC*. IEEE, 2002.

[117] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster, “The anatomy of a context-aware application,” in *Wireless Networks*. Springer-, 2002.

[118] A. Bensky, *Wireless positioning technologies and applications*. Artech House, 2007.

[119] P. Rong and M. L. Sichitiu, “Angle of arrival localization for wireless sensor networks,” in *IEEE Communications Society on Sensor and Ad Hoc Communications and Networks (SECON)*, vol. 1. IEEE, 2006, pp. 374–382.

[120] P. Kułakowski, J. Vales-Alonso, E. Egea-López, W. Ludwin, and J. García-Haro, “Angle-of-arrival localization based on antenna arrays for wireless sensor networks,” in *Computers & Electrical Engineering*, vol. 36, no. 6. Elsevier, 2010, pp. 1181–1186.

[121] “Design Compiler,” http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler.

[122] B. I. C. R. Group, “libscapi,” https://github.com/cryptobiu/libscapi, 2017.

[123] “Intel Atom Processor E3845,” ark.intel.com/products/78475, 2015.

[124] “IEEE 1609 - Family of Standards for Wireless Access in Vehicular Environments (WAVE),” standards.its.dot.gov/factsheets/factsheet/80, 2009.

[125] F. Armknecht, R. Maes, A.-R. Sadeghi, B. Sunar, and P. Tuyls, “Memory leakage-resilient encryption based on physically unclonable functions,” in *Towards Hardware-Intrinsic Security*. Springer, 2010.

[126] S. P. Skorobogatov, “Semi-invasive attacks: a new approach to hardware security analysis,” University of Cambridge, Tech. Rep., 2005.

[127] L. Kulseng, Z. Yu, Y. Wei, and Y. Guan, “Lightweight mutual authentication and ownership transfer for RFID systems,” in *INFOCOM*. IEEE, 2010.

[128] S. Katzenbeisser, Ü. Kocabaş, V. Van Der Leest, A.-R. Sadeghi, G.-J. Schrijen, and C. Wachsmann, “Recyclable PUFs: logically reconfigurable PUFs,” *Journal of Cryptographic Engineering*, vol. 1, no. 3, p. 177, 2011.

[129] A.-R. Sadeghi, I. Visconti, and C. Wachsmann, “Enhancing RFID security and privacy by physically unclonable functions,” in *Towards Hardware-Intrinsic Security*. Springer, 2010.

[130] M. Majzoobi, M. Rostami, F. Koushanfar, D. S. Wallach, and S. Devadas, "Slender PUF protocol: A lightweight, robust, and secure authentication by substring matching," in *Symposium on Security and Privacy Workshops*. IEEE, 2012.

[131] D. Moriyama, S. Matsuo, and M. Yung, "PUF-Based RFID Authentication Secure and Private under Complete Memory Leakage." *IACR Cryptology ePrint Archive*, 2013.

[132] A. Aysu, E. Gulcan, D. Moriyama, P. Schaumont, and M. Yung, "End-to-end design of a PUF-based privacy preserving authentication protocol," in *CHES*. Springer, 2015.

[133] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "FPGA intrinsic PUFs and their use for IP protection," in *CHES*. Springer, 2007.

[134] C. Herder, M.-D. Yu, F. Koushanfar, and S. Devadas, "Physical Unclonable Functions and applications: A tutorial," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1126–1141, 2014.

[135] W. Xiong, A. Schaller, N. A. Anagnostopoulos, M. U. Saleem, S. Gabmeyer, S. Katzenbeisser, and J. Szefer, "Run-time Accessible DRAM PUFs in Commodity Devices," in *CHES*. Springer, 2016.

[136] A. Van Herrewege, S. Katzenbeisser, R. Maes, R. Peeters, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann, "Reverse fuzzy extractors: Enabling lightweight mutual authentication for PUF-enabled RFIDs," in *International Conference on Financial Cryptography and Data Security*. Springer, 2012.

[137] C. Herder, L. Ren, M. van Dijk, M.-D. Yu, and S. Devadas, "Trapdoor computational fuzzy extractors and stateless cryptographically-secure physical unclonable functions," *IEEE Transactions on Dependable and Secure Computing*, vol. 14, no. 1, 2017.

[138] A. Afshar, P. Mohassel, B. Pinkas, and B. Riva, "Non-interactive secure computation based on cut-and-choose," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2014.

[139] L. T. Brandão, "Secure two-party computation with reusable bit-commitments, via a cut-and-choose with forge-and-lose technique," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2013.

[140] X. Wang, A. J. Malozemoff, and J. Katz, "Faster secure two-party computation in the single-execution setting," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2017.

[141] P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," in *STOC*, Dallas, TX, 1998.

[142] P. Indyk and D. Woodruff, "Polylogarithmic private approximations and efficient matching," in *Theory of Cryptography*. Springer, 2006.

[143] Y. Dodis, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," in *EUROCRYPT*. Springer, 2004.

[144] U. Ruhrmair, S. Devadas, and F. Koushanfar, "Security based on Physical Unclonability and Disorder," *Introduction to Hardware Security and Trust*, 2011.

[145] R. Maes, V. Rozic, I. Verbauwhede, P. Koeberl, E. Van der Sluis, and V. van der Leest, "Experimental evaluation of Physically Unclonable Functions in 65 nm CMOS," in *ESSCIRC*. IEEE, 2012.

[146] F. Kohnhäuser, A. Schaller, and S. Katzenbeisser, "PUF-based software protection for low-end embedded devices," in *ICTTC*. Springer, 2015.

[147] G.-J. Schrijen and V. van der Leest, "Comparative analysis of SRAM memories used as PUF primitives," in *DATE*. EDA Consortium, 2012.

[148] C. Keller, F. Gurkaynak, H. Kaeslin, and N. Felber, "Dynamic memory-based physically unclonable function for the generation of unique identifiers and true random numbers," in *ISCAS*. IEEE, 2014.

[149] S. Rosenblatt, S. Chellappa, A. Cestero, N. Robson, T. Kirihata, and S. S. Iyer, "A self-authenticating chip architecture using an intrinsic fingerprint of embedded DRAM," *Journal of Solid-State Circuits*, vol. 48, no. 11, pp. 2934–2943, 2013.

[150] J. Delvaux, R. Peeters, D. Gu, and I. Verbauwhede, "A survey on lightweight entity authentication with strong PUFs," *Computing Surveys*, 2015.

[151] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich, "Scifi-a system for secure face identification," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010.

[152] J. Bringer, H. Chabanne, and B. Kindarji, "Identification with encrypted biometric data," *Security and Communication Networks*, vol. 4, no. 5, 2011.

[153] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft, "Privacy-preserving face recognition," in *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 2009.

[154] M. S. Riazi, N. K. Dantu, L. V. Gattu, and F. Koushanfar, "GenMatch: Secure DNA compatibility testing," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2016, pp. 248–253.

[155] A. Jarrous and B. Pinkas, "Secure hamming distance based computation and its applications," in *International Conference on Applied Cryptography and Network Security*. Springer, 2009.

[156] A. Z. Broder, "On the Resemblance and Containment of Documents," in *the Compression and Complexity of Sequences*, Positano, Italy, 1997.

[157] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, “Min-Wise Independent Permutations,” in *STOC*, Dallas, TX, 1998.

[158] ——, “Min-wise independent permutations,” *Journal of Computer and System Sciences*, vol. 60, no. 3, pp. 630–659, 2000.

[159] J. L. Carter and M. N. Wegman, “Universal classes of hash functions,” in *STOC*, 1977.

[160] G. P. G. P. Wadsworth and J. G. Bryan, *Introduction to probability and random variables*. McGraw-Hill, New York, 1960.

[161] S. J. Haberman, “Discrete Multivariate Analysis: Theory and Practice,” 1976.

[162] O. Goldreich, *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[163] R. Canetti, “Security and composition of multiparty cryptographic protocols,” *Journal of Cryptology*, vol. 13, no. 1, 2000.

[164] C. Wolf, “Yosys Open SYnthesis Suite,” http://www.clifford.at/yosys/.

[165] A. Shrivastava, “Optimal Densification for Fast and Accurate Minwise Hashing,” *arXiv preprint arXiv:1703.04664*, 2017.

[166] A. Shrivastava and P. Li, “Densifying One Permutation Hashing via Rotation for Fast Near Neighbor Search.” in *ICML*, 2014.

[167] M. S. Riazi, B. Chen, A. Shrivastava, D. Wallach, and F. Koushanfar, “Sub-linear Privacy-preserving Search with Untrusted Server and Semi-honest Parties,” *arXiv preprint arXiv:1612.01835*, 2016.

[168] V. Kolesnikov, A. Sadeghi, and T. Schneider, “Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima,” in *CANS*. Springer, 2009.

[169] M. Shaneck, Y. Kim, and V. Kumar, “Privacy Preserving Nearest Neighbor Search,” in *ICDMW*. Springer, 2006.

[170] Y. Qi and M. J. Atallah, “Efficient Privacy-Preserving k-Nearest Neighbor Search,” in *ICDCS*. IEEE, 2008, pp. 311–319.

[171] Y. Lindell and B. Pinkas, “Privacy preserving data mining,” *Journal of cryptology*, vol. 15, no. 3, 2002.

[172] ——, “A Proof of Yao’s Protocol for Secure Two-Party Computation,” *Journal of Cryptology*, 2009.

[173] Y. Huang, D. Evans, and J. Katz, “Private set intersection: Are garbled circuits better than custom protocols?” in *NDSS*, 2012.

[174] A. Waksman, "A permutation network," *Journal of the ACM (JACM)*, vol. 15, no. 1, pp. 159–163, 1968.

[175] E. De Cristofaro and G. Tsudik, "Practical Private Set Intersection Protocols with Linear Complexity." in *Financial Cryptography*, vol. 10. Springer, 2010, pp. 143–159.

[176] B. Pinkas, T. Schneider, G. Segev, and M. Zohner, "Phasing: Private Set Intersection Using Permutation-based Hashing." in *USENIX Security Symposium*, 2015, pp. 515–530.

[177] V. Kolesnikov, N. Matania, B. Pinkas, M. Rosulek, and N. Trieu, "Practical Multi-party Private Set Intersection from Symmetric-Key Techniques," in *CCS*. ACM, 2017.

[178] V. Kepuska and G. Bohouta, "Next-generation of virtual personal assistants (microsoft cortana, apple siri, amazon alexa and google home)," in *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2018, pp. 99–103.

[179] I. Masi, Y. Wu, T. Hassner, and P. Natarajan, "Deep face recognition: A survey," in *2018 31st SIBGRAPI conference on graphics, patterns and images (SIBGRAPI)*. IEEE, 2018, pp. 471–478.

[180] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean, "A guide to deep learning in healthcare," *Nature medicine*, vol. 25, no. 1, p. 24, 2019.

[181] A. Esteva, B. Kuprel, R. A. Novoa, J. Ko, S. M. Swetter, H. M. Blau, and S. Thrun, "Dermatologist-level classification of skin cancer with deep neural networks," *Nature*, vol. 542, no. 7639, p. 115, 2017.

[182] B. Alipanahi, A. Delong, M. T. Weirauch, and B. J. Frey, "Predicting the sequence specificities of DNA-and RNA-binding proteins by deep learning," *Nature biotechnology*, vol. 33, no. 8, p. 831, 2015.

[183] A. Rajkomar, E. Oren, K. Chen, A. M. Dai, N. Hajaj, M. Hardt, P. J. Liu, X. Liu, J. Marcus, M. Sun *et al.*, "Scalable and accurate deep learning with electronic health records," *npj Digital Medicine*, vol. 1, no. 1, p. 18, 2018.

[184] A. Alameen and A. Gupta, "Optimization driven deep learning approach for health monitoring and risk assessment in wireless body sensor networks," *International Journal of Business Data Communications and Networking (IJBDCN)*, vol. 16, no. 1, pp. 70–93, 2020.

[185] L. Xie and A. Yuille, "Genetic cnn," *arXiv preprint arXiv:1703.01513*, 2017.

[186] Z. Ghodsi, A. Veldanda, B. Reagen, and S. Garg, "Cryptonas: Private inference on a relu budget," in *Advances in Neural Information Processing Systems*, 2020.

[187] 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*. ACM, 2017, pp. 619–631.

[188] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *27th ${$USENIX$}$ Security Symposium (${$USENIX$}$ Security 18)*, 2018, pp. 1651–1669.

[189] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved primitives for MPC over mixed arithmetic-binary circuits," in *Annual International Cryptology Conference*. Springer, 2020, pp. 823–852.

[190] N. Agrawal, A. Shahin Shamsabadi, M. J. Kusner, and A. Gascón, "QUOTIENT: two-party secure neural network training and prediction," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1231–1247.

[191] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[192] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.

[193] E. Hesamifard, H. Takabi, and M. Ghasemi, "Cryptodl: Deep neural networks over encrypted data," *arXiv preprint arXiv:1711.05189*, 2017.

[194] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *Annual International Cryptology Conference*. Springer, 2018, pp. 483–512.

[195] E. Chou, J. Beal, D. Levy, S. Yeung, A. Haque, and L. Fei-Fei, "Faster cryptonets: Leveraging sparsity for real-world encrypted inference," *arXiv preprint arXiv:1811.09953*, 2018.

[196] A. Sanyal, M. Kusner, A. Gascon, and V. Kanade, "TAPAS: Tricks to accelerate (encrypted) prediction as a service," in *International Conference on Machine Learning*. PMLR, 2018, pp. 4490–4499.

[197] B. D. Rouhani, M. S. Riazi, and F. Koushanfar, "DeepSecure: Scalable Provably-Secure Deep Learning," *arXiv preprint arXiv:1705.08963*, 2017.

[198] M. Ball, B. Carmer, T. Malkin, M. Rosulek, and N. Schimanski, "Garbled Neural Networks are Practical." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 338, 2019.

[199] S. Bian, M. Hiromoto, and T. Sato, "DArL: Dynamic parameter adjustment for LWE-based secure inference," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1739–1744.

[200] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *25th ${$USENIX$}$ Security Symposium (${$USENIX$}$ Security 16)*, 2016, pp. 601–618.

[201] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *ACM CCS*, 2015.

[202] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 3–18.

[203] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[204] M. Abadi, A. Chu, I. Goodfellow, H. B. McMahan, I. Mironov, K. Talwar, and L. Zhang, "Deep Learning with Differential Privacy," *arXiv preprint arXiv:1607.00133*, 2016.

[205] Z. Yang, B. Shao, B. Xuan, E.-C. Chang, and F. Zhang, "Defending model inversion and membership inference attacks via prediction purification," *arXiv preprint arXiv:2005.03915*, 2020.

[206] Q. Chen, C. Xiang, M. Xue, B. Li, N. Borisov, D. Kaarfar, and H. Zhu, "Differentially private data generative models," *arXiv preprint arXiv:1812.02274*, 2018.

[207] M. Samragh, M. Ghasemzadeh, and F. Koushanfar, "Customizing neural networks for efficient fpga implementation," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 85–92.

[208] M. S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing, "Looknn: Neural network with no multiplication," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1775–1780.

[209] S. U. Hussain, M. Javaheripi, M. Samragh, and F. Koushanfar, "COINN: Crypto/ML Codesign for Oblivious Inference via Neural Networks," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[210] P. Rindal, "libOTe: an efficient, portable, and easy to use Oblivious Transfer Library," https://github.com/osu-crypto/libOTe.

[211] F. Tramer and D. Boneh, "Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware," in *International Conference on Learning Representations*, 2019.

[212] N. Kumar, M. Rathee, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow: Secure tensorflow inference," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 336–353.

[213] J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang, "Slimmable neural networks," *arXiv preprint arXiv:1812.08928*, 2018.

[214] L. Liu and J. Deng, "Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[215] G. Guennebaud, B. Jacob *et al.*, “Eigen v3,” http://eigen.tuxfamily.org, 2010.

[216] M. Javaheripi, M. Samragh, T. Javidi, and F. Koushanfar, “GeneCAI: gene tic evolution for acquiring c ompact AI,” in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, 2020, pp. 350–358.

[217] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, “Chameleon: A hybrid secure computation framework for machine learning applications,” in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 707–721.

[218] N. Chandran, D. Gupta, A. Rastogi, R. Sharma, and S. Tripathi, “EzPC: programmable, efficient, and scalable secure two-party computation for machine learning,” *ePrint Report*, vol. 1109, 2017.

[219] S. Truex, N. Baracaldo, A. Anwar, T. Steinke, H. Ludwig, R. Zhang, and Y. Zhou, “A hybrid approach to privacy-preserving federated learning,” in *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, 2019, pp. 1–11.

[220] R. Xu, N. Baracaldo, Y. Zhou, A. Anwar, and H. Ludwig, “Hybridalpha: An efficient approach for privacy-preserving federated learning,” in *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*, 2019, pp. 13–23.

[221] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl, “Efficient two-round OT extension and silent non-interactive secure computation,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 291–308.

[222] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl, “Efficient pseudorandom correlation generators: Silent OT extension and more,” in *Annual International Cryptology Conference*. Springer, 2019, pp. 489–518.