

**Efficient Constructions and Implementations for
Secure Multi-Party Computation**

KAZUMA OHARA

THE UNIVERSITY OF ELECTRO-COMMUNICATIONS
SEPTEMBER 2019

Efficient Constructions and Implementations for Secure Multi-Party Computation

THE UNIVERSITY OF ELECTRO-COMMUNICATIONS
GRADUATE SCHOOL OF INFORMATICS AND ENGINEERING
A DISSERTATION SUBMITTED FOR
DOCTOR OF PHILOSOPHY IN ENGINEERING

By
Kazuma Ohara
September, 2019

Efficient Constructions and Implementations for Secure Multi-Party Computation

Supervisory Committee

Chairperson: Mitsugu Iwamoto

Member: Kazuo Ohta

Member: Kazuo Sakiyama

Member: Hiroshi Yoshiura

Member: Yasutada Oohama

© Copyright 2019
by
Kazuma Ohara

Abstract

Multi-Party Computation (MPC) enables a set of parties to compute joint functions on their private inputs while concealing the parties' private inputs. In recent years, privacy is recognized as an important issue in the field of handling personal information such as cloud computing, data mining, machine learning, etc. MPC is attracting attention as a method that can solve this issue and thus the practical realization of MPC is desired as soon as possible.

We approach efficient realization method for MPC that meet practical performance in the real-world applications from both theory and implementation. In this thesis, we aim to provide efficient MPC protocols design and implementation achieving high-throughput. Since MPC requires communication among multiple parties, the efficiency of MPC is physically limited by the performance of the communication bandwidth. Therefore, we explore the optimal method by improvement of computation and communication cost, particularly communication.

The results in this thesis are as follows:

- **Foundation (1): High-throughput semi-honest secure 3-party computation based on replicated SS with honest-majority:** In this thesis, we describe a new information-theoretic protocol (and a computationally-secure variant) for secure three-party computation with an honest majority. The protocol has very minimal computation and communication; for Boolean circuits, each party sends only a single bit for every AND gate (and nothing is sent for XOR gates). We demonstrate the practical potential of our protocol by implementing a MPC-based system for Kerberos authentication, which is a well-known network authentication protocol based on symmetric-key cryptosystems.
- **Foundation (2): Optimizing cheating detection for honest-majority**

MPC: We provide general techniques for improving efficiency of cut-and-choose protocols on multiplication triples and utilize them to significantly improve the recently published protocol of Furukawa et al. Most notably, we design cache-efficient shuffling techniques for implementing cut-and-choose without randomly permuting large arrays (which is very slow due to continual cache misses). We provide a combinatorial analysis of our techniques, bounding the cheating probability of the adversary. Our results demonstrate that high-throughput secure computation for malicious adversaries is possible.

- **Application (1): Compiler for SS-based MPCs:** Today, we have protocols that can carry out large and complex computations in very reasonable time (and can even be very fast, depending on the computation and the setting). Despite this amazing progress, there is still a major obstacle to the adoption and use of MPC due to the huge expertise needed to design a specific MPC execution.

In this thesis, we design and implement a MPC compiler for our three-party honest majority MPC. Our implementation is an extension of a well-known MPC compiler called “SPDZ compiler” so that it can work with general underlying protocols. In this thesis we called the compiler we made “generalized SPDZ compiler”. Moreover, our SPDZ extensions were made in mind to enable the use of SPDZ for arbitrary protocols and to make it easy for others to integrate existing and new protocols.

- **Application (2): Dedicated MPC protocols for high-level functionalities** Although our SS-based 3-party MPC proposed in the above results is very efficient in general, the SS-based MPCs are still inefficient for several heavy computations like algebraic operations, as they require a large amount and number of communication proportional to the number of multiplications in the operations (which is not the case with other SS-based MPCs). In this thesis, we propose the following two dedicated MPC protocols for high-level functionalities; (1) Arithmetic-to-Boolean/Boolean-to-Arithmetic conversion and (2) modular exponentiation, to accelerate SS-based MPC further.

Abstract (in Japanese)

「マルチパーティ計算 (Multi-Party Computation; MPC)」は、複数の参加者がその個人情報をも他の参加者に秘匿したまま、それらを入力とする関数の計算を参加者間の協調計算で行う暗号プロトコルの一種である。近年のクラウドコンピューティングやデータマイニング、機械学習などの個人情報を扱う分野で、データの取り扱いに関するプライバシーの問題が課題として認識される中で、MPCはこの課題を解決し得る手法として注目されており、一刻も早い実用化が求められている。

理論的には、MPCは任意の関数を計算可能であることが知られている。しかしながら、MPCの実用化にあたってはその効率が大きな課題となっている。30年以上研究がなされた現在においてもなお、MPCは実社会が要求する機能・性能を達成できているとは言い難い。

実システムにおいては、処理の応答の速さ（レイテンシ）や、単位時間当たりの処理件数（スループット）が求められる。特に、データ分析（データマイニングなど）のユースケースでは、リアルタイム処理は必ずしも重要ではないため、スループットがより重要視される。しかしながら、MPCは参加者間での通信を必要とするため、その効率は通信路の物理的性能によって制限される。従来のMPC技術は、その通信量の多さに由来するスループットの限界を抱えていた。

本研究の目的は、MPCの高速な実現方法について理論・実装の両面からアプローチし、高スループットなMPCプロトコルの設計・実装を提供することにある。

本論文では、得られた成果は以下の通りである。

[Part I（基礎）：任意の回路を計算するより効率的な3射MPCフレームワーク]

- I-(1): 受動的攻撃者に対して安全な低通信量3者間MPCフレームワークの提案（第3章）本研究では、複製型秘密分散法（Replicated Secret Sharing Schemes; RSSS）を用いた情報理論的安全性を持つ3者間MPC方式とその高速実装を提案した。また、Kerberos認証と呼ばれる共通鍵ベースの認証プロトコルをMPCで行うシステムを実装し、SSSベースのMPCが現実的な性能要件を満たし得

るスループットを実現できることを示した。

- I-(2): 能動的な攻撃者に対する MPC の不正検知手法の効率改善（第 4 章）(1) で提案したプロトコルは，受動的な攻撃者に対してのみ安全な MPC プロトコルである．本研究では，(1) の MPC における能動的な攻撃者を検知する手法を併せて提案する．提案手法は，既存手法の通信量を理論的改善によって 30%削減した．また，本プロトコルにおける計算量的に高コストな部分を改善するための手法を提案し，実装評価によってこの手法の効果を実証した．

[Part 2（応用）：より複雑な関数を効率的に実現する手法の提案]

- II-(1): MPC のプログラムを自動生成する「MPC コンパイラ」の設計と実装（第 5 章）ある関数を MPC で実行するには，その関数を MPC で実現可能な低レベルのコンポーネントで表現する必要があるが，これを人手で行うことは複雑な関数においては現実的ではない．また，MPC 専用のプログラム記述には MPC に対する知識を必要とする．これを解決するため，MPC の研究ではしばしば，専用のプログラム記述から秘密計算の実行形式を自動生成する「MPC コンパイラ」がプロトコルと合わせて提案される．本論文では(1)のフレームワークのための専用命令セットをサポートする MPC コンパイラを設計・実装し，本論文で提案する効率的な MPC を現実のさまざまな問題に適用するためのツールとして提供する．
- II-(2): より高機能な関数のための専用 MPC プロトコル（第 6 章）(1)(2) のフレームワークは，加算や乗算などの低レベルなコンポーネントを組み合わせることで任意の関数計算を実現するが，より複雑な関数を実現したい場合には専用の「MPC モジュール」を設計することによって更なる効率化を実現し得る．本論文では，利用頻度の高い関数として，論理型・算術型の値の相互変換を行う「型変換」，および暗号系処理などで特に用いられる「べき乗剰余関数」のより効率的な MPC プロトコルを提案し，実装によってこのアプローチの有効性を実証した．

Acknowledgments

I would like to express my sincere thanks to my supervisor, Assoc. Prof. Mitsugu Iwamoto and Prof. Kazuo Ohta for their valuable input, feedbacks and discussions. Their insightful comments always made me aware of my deficiencies and improved my dissertation and presentation. Their enthusiasm and great mathematical skills have been an inspiration throughout the thesis. It has been an absolute pleasure working with and learning from them. I owe what I am today to their education.

I am also indebted to the members of the supervisory committee, Prof. Kazuo Sakiyama, Prof. Hiroshi Yoshiura and Prof. Yasutada Oohama. Their all valuable comments and discussions gave me a lot of perception and was helpful for improve the thesis.

I am very thankful to NEC and to all my colleagues for their kind support. Especially, I would like to acknowledge co-authors, Dr. Jun Furukawa, Dr. Toshinori Araki, Mr. Hikaru Tsuchida. Since I was joined NEC, Dr. Jun Furukawa and Dr. Toshinori Araki helped me very well as my supervisors and as colleagues. Mr. Tsuchida joined from the middle of our project and support us strongly. I also would like to acknowledge Mr. Koichi Konishi and Dr. Takao Takenouchi, who are managers of our team, for their understanding, support and all effort on business development for our research.

I would like to express my gratitude to the co-authors, Prof. Yehuda Lindell, Mr. Assi Barak, Mr. Ariel Nof, Ms. Adi Watzman, Mr. Tamar Lichter, Mr. Or Weinstein in Bar-Ilan University, and Marcel Keller in Data 61. I believe they are the world's leading teams in the area, and I have learned a lot of things from them. Especially, Prof. Yehuda Lindell showed me how to write papers, and how to manage a research project. and more. Mr. Assi Barak showed me a lot of techniques for software design and engineering. Being able to see their work up close led to my great growth. In addition, I also thank to the all member of Bar-Ilan University for giving

a lot of great time during my long stay in Israel. In particular, I am highly thankful to Prof. Benny Pinkas for interesting conversations for researches and more, and to Ms. Yonit Yonit Homburger for all kind support on my stay. The time I spent with them was a lot of fun and became a great asset to me.

I am deeply grateful to Dr. Yohei Watanabe in National Institute of Information and Communications Technology (previously in University of Electro-Communications) for a lot of discussion on the state-of-the-art research and daily conversation. On the result of Chapter 6, I receive a lot of valuable comments from him and it make the paper better quality. However, the most important thing is that the time of discussion with him was very enjoyable and gave me a lot of fun and stimulation in my doctoral course.

Finally, it is my pleasure to thank my father, mother and sister for their supporting and encouraging me for my education.

September, 2019
Kazuma Ohara
Tokyo, Japan

List of Figures and Tables

Chapter 1

- Table 1.1: Known Feasibilities on MPC

Chapter 3

- Table 3.1: Reported times for semi-honest 3-party computation & honest majority; the throughput is measured in AES computations per second (the last two rows with similar configurations)
- Table 3.2: Experiment results running AES-CTR. The CPU column shows the average CPU utilization per core, and the network column is in Gbps per server. Latency is given in milliseconds
- Figure 3.1: Bit-slice operation
- Figure 3.2: Unpack operation of AVX instruction set
- Figure 3.3: Moving masked bit operation of AVX instruction set
- Figure 3.4: Throughput per core (AES computations)
- Figure 3.5: Latency versus throughput (AES)
- Figure 3.6: The Kerberos authentication using MPC

Chapter 4

- Table 4.1: Implementation results; throughput
- Table 4.2: Implementation results; B denotes the bucket size; security level 2^{-40}

- Figure 4.1: Microbenchmarking of the baseline implementation (the protocol of [56]), using the CxxProf C++ profiler
- Figure 4.2: Cache-efficient shuffling method
- Figure 4.3: Microbenchmarking of Protocol 4.4, using the CxxProf C++ profiler
- Figure 4.4: Architecture of implementation
- Figure 4.5: Microbenchmarking of best protocol variant, using the CxxProf C++ profiler (run on a local host)

Chapter 5

- Table 5.1: Decision tree computation (seconds).
- Table 5.2: Running times for batch vectorization in seconds. $\text{Batch} \times N$ means running N executions in parallel (i.e., with vectors of length N).
- Figure 5.1: SPDZ Python code for oblivious selection from an array
- Figure 5.2: High-level SPDZ compiler architecture
- Figure 5.3: Representing a program as a directed acyclic graph
- Figure 5.4: Multiplication in the original SPDZ compiler vs using new instruction extension
- Figure 5.5: The extensions applied to the SPDZ compiler of [42]
- Figure 5.6: Benchmarking on Mean computation (X-axis=num. inputs)
- Figure 5.7: Benchmarking on Variance computation (X-axis=num. inputs)
- Figure 5.7: Benchmarking on Variance computation (X-axis=num. inputs)
- Figure 5.8: Benchmarking on US Census SQL query (X-axis=num. inputs)

Chapter 6

- Table 6.1: Reference for local re-sharing for bit-decomposition
- Table 6.2: Truth table for computing x^j via majority
- Table 6.3: Truth table for checking correctness on x^j
- Table 6.4: Reference for local re-sharing for ring composition
- Table 6.5: Different parameters and their cost
- Table 6.6: Optimal block-size and costs for the variable-length approach (computation is from right-to-left)
- Table 6.7: Costs for the conditional-sum adder approach
- Table 6.8: Complexity of decomposition and ring composition
- Table 6.9: Comparison of Complexity of 32-bit Integer Conversions for Secure 3-Party Computation
- Table 6.10: Complexity of MPC for Modular Exponentiation over Replicated Secret Sharing
- Table 6.11: Appropriate data length of discrete-log based cryptosystem for 128/256-bit security
- Figure 6.1: Comparison for communication bits between previous scheme and scheme 1 in this paper
- Figure 6.2: Latency-field size with 128-bit security parameter ($\log q = 256$ bit)
- Figure 6.3: Latency-field size with 256-bit security parameter ($\log q = 512$ bit)

List of Functionalities

Chapter 3

- Functionality 3.1: $\mathcal{F}_{\text{MULT}}$ – multiplication
- Functionality 3.2: \mathcal{F}_{CR} – corr. randomness

Chapter 6

- Functionality 6.1: \mathcal{F}_{mpc} – The Mixed MPC Functionality

List of Protocols

Chapter 3

- Protocol 3.1: Sharing input to the parties
- Protocol 3.2: Computing XOR gate
- Protocol 3.3: Computing AND gate
- Protocol 3.4: Generating computational correlated randomness
- Protocol 3.5: Computing arithmetic addition gate over \mathbb{Z}_q
- Protocol 3.6: Computing arithmetic multiplication gate
- Protocol 3.7: Computing $\mathcal{F}_{\text{MULT}}$
- Protocol 3.8: Computing \mathcal{F}_{CR}

Chapter 4

- Protocol 4.1: Computing a function f with Malicious Adversaries
- Protocol 4.2: Generating Valid Triples – Cache-Efficiently
- Protocol 4.3: Computing f with Malicious Adversaries – Smaller Buckets
- Protocol 4.4: Computing f with Malicious Adversaries – On-Demand Shuffling and Smaller Buckets

Chapter 6

- Protocol 6.1: Communication-Efficient Bit Decomposition from \mathbb{Z}_{2^n} to $(\mathbb{Z}_2)^n$
- Protocol 6.2: Communication-Efficient Ring Composition from $(\mathbb{Z}_2)^n$ to \mathbb{Z}_{2^n}

- Protocol 6.3: Previous work: modular exponentiation with public base
- Protocol 6.4: Skew Exponentiation
- Protocol 6.5: Modular Exponentiation with prime modulus
- Protocol 6.6: Modular Exponentiation with the case where modulus is power of 2
- Protocol 6.7: Modular Exponentiation in the special case where $p = 2q + 1$
- Protocol 6.8: Modular Exponentiation in the special case where $p > 3q + 1$

List of Notation

Set notation

- \mathbb{Z} : a set of integer
- \mathbb{N} : a set of the natural number (including 0)
- \mathbb{R} : a set of the real number
- $\mathbb{Z}_q = \{0, 1, \dots, q-1\}$: a quotient ring with positive integer q
- $\{0, 1\}^k$: a set of bit strings with length k
- $\{0, 1\}^* := \cup_{k \in \mathbb{N}} \{0, 1\}^k$
- $[0, 1]$: a closed interval; $\forall x \in [0, 1], 0 \leq x \leq 1$.

Miscellaneous symbol

- \equiv : perfectly indistinguishable
- $\stackrel{s}{\equiv}$: statistically indistinguishable
- $\stackrel{c}{\equiv}$: computationally indistinguishable
- κ : security parameter

MPC parties notation

- \mathcal{P} : a set of parties
- $P_i \in \mathcal{P}$: a party with the identifier i (for 3-party case, $i \in \{0, 1, 2\}$)

Representation on secret sharing and MPC

- $\pi \equiv f$: the real protocol π securely computes a functionality f (for the definition of “securely compute”, see also Definition 2.4.3)
- $\pi^g \equiv f$: the real protocol π securely computes a functionality f in the g -hybrid model
- $[x]^q$: x is shared among the parties by a (linear) secret sharing scheme with modulus q .
- $[x]_i^q$: P_i 's share for the secret x .
- $[x]_{i,j}^q$: j -th element of P_i 's share for the secret x . Namely, if P_i 's share is a vector of length m , $[x]_i^q = ([x]_i^q, 0, \dots, [x]_{i,m-1}^q)$.
 x_0, x_1, x_2 : sub-shares of $[x]^q$ in the case of replicated secret sharing ($x_0 + x_1 + x_2 \bmod q = x$).
- x^j : j -th significant bit of x .
- $[a]_i^q || [b]_i^q$: concatenation of P_i 's shares $[a]_i^q$ and $[b]_i^q$. Namely, $([a]^{q,i} || [b]^{q,i}) = ([a]_{i,0}^q || [b]_{i,0}^q, [a]_{i,1}^q || [b]_{i,1}^q, \dots, [a]_{i,m-1}^q || [b]_{i,m-1}^q)$.

MPC protocols

- **add**: addition (if Boolean circuit, it implies XOR)
- **mult**: multiplication (if Boolean circuit, it implies AND)
- **bit_decomp**: bit decomposition
- **skew_decomp**: skew decomposition
- **ring_comp**: ring composition
- **pub_expo**: modular exponentiation with public base (previous work)
- **skew_expo**: skew exponentiation
- **modexp_p**: modular exponentiation with public base and prime modulus

- `modexp2n`: modular exponentiation with public base and the modulus of 2 powers
- `modexpsp`: modular exponentiation with public base and $p = 2q + 1$
- `modexpsp2`: modular exponentiation with public base and $p = 3q + 1$

Contents

Abstract	v
Abstract (in Japanese)	vii
Acknowledgments	ix
List of Figures and Tables	xi
List of Functionalities	xiv
List of Notation	xvii
1 Introduction	1
1.1 Theoretical Background on Secure Multi-Party Computation	1
1.2 Practical Background on Secure Multi-Party Computation	9
1.3 Motivation and Our Results	11
1.4 Organization of the Thesis	15
2 Preliminaries	17
2.1 Notation	17
2.2 Indistinguishability	18
2.3 Secret Sharing	19
2.3.1 (k, n) -threshold schemes	19
2.3.2 Replicated Secret Sharing	20
2.4 The Model of Secure Computation	20
2.4.1 Settings	21
2.4.2 Security Criteria	21
2.4.3 Simulation-based Security	23

2.4.4	Representation of Functionalities for Secret Sharing-based 3PC	26
-------	--	----

I Foundations: Secure 3-Party Computation for General Circuits — Theory and Implementations for More Efficient Primitives **27**

3	Foundation (1): Semi-Honest Secure 3-Party Computation based on Replicated Secret Sharing	28
3.1	Introduction	28
3.2	Related Work	30
3.3	The New Communication-Efficient Protocol for 3-Party Computation	31
3.3.1	Securely Computing Boolean Circuits	32
3.3.2	Generating Correlated Randomness	35
3.3.3	The Ring with General Modulus: 2^n and Fields	36
3.3.4	Protocol Efficiency and Comparison	38
3.4	Security against Semi-Honest Adversaries	39
3.4.1	Computing f in the $\mathcal{F}_{\text{MULT}}$ -Hybrid Model	40
3.4.2	Computing $\mathcal{F}_{\text{MULT}}$ in the \mathcal{F}_{CR} -Hybrid Model	46
3.4.3	Computing \mathcal{F}_{CR} in the Plain Model	49
3.4.4	Wrapping Up	51
3.5	Security against Malicious Adversaries in the Client-Server Model . .	52
3.6	Experimental Evaluation	54
3.6.1	Implementation Aspects	54
3.6.2	Result (1): Fast AES	56
3.6.3	Result (2): Kerberos KDC with Shared Passwords	58
4	Foundation (2): Maliciously Secure 3-Party Computation based on Replicated Secret Sharing	62
4.1	Introduction	62
4.2	Related Work	66
4.3	The Baseline Protocol	67
4.3.1	An Informal Description	67
4.3.2	Implementation Results and Needed Optimizations	70

4.4	Optimized Cheater Identification	72
4.4.1	Cache-Efficient Shuffling for Cut-and-Choose	72
4.4.2	Reducing Bucket-Size and Communication	80
4.4.3	Smaller Buckets With On-Demand Secure Computation	85
4.4.4	Hash Function Optimization	92
4.5	Trade-off between Security and Efficiency: The Combinatorics of Cut-and-Choose	93
4.5.1	The Potential of Different-Sized Buckets	94
4.5.2	Moderately Lowering the Cheating Probability	96
4.6	Experimental Evaluation	99
4.6.1	Implementation Aspects	99
4.6.2	Results and Discussion	102

II Applications: How to Realize MPCs for Complex Functionalities — Bridging from Efficient Primitives to Efficient Applications **104**

5	Application (1): Generalized SPDZ Compiler for MPC based on Secret Sharing	105
5.1	Introduction	105
5.2	Related Work	105
5.3	Review on the SPDZ Protocol and Compiler	109
5.3.1	Overview	109
5.3.2	Circuit Optimizations	110
5.3.3	Higher-Level Algorithms	112
5.4	Software Design and Implementation for Making SPDZ a General Compiler	113
5.4.1	Modifications to the SPDZ Compiler	114
5.4.2	Incorporating BMR Circuits	118
5.5	Experimental Evaluation	119
5.5.1	Implementation Aspects	119
5.5.2	Results and Discussion	120

6	Application (2): 3-Party Computation for High-Level Functions	126
6.1	MPC for Bit Decomposition and Ring Composition	126
6.1.1	Introduction	126
6.1.2	Related Work	128
6.1.3	Communication-Efficient Bit Decomposition	129
6.1.4	Communication-Efficient Ring Composition	133
6.1.5	Variants: Reducing the Round Complexity	136
6.1.6	Security	140
6.1.7	Efficiency	142
6.2	MPC for Exponentiation	144
6.2.1	Introduction	144
6.2.2	Related Work	146
6.2.3	A Key Technique: Skew Exponentiation	147
6.2.4	Communication-Efficient Modular Exponentiation	149
6.2.5	Security	153
6.2.6	Efficiency	154
6.3	Experimental Evaluation	156
6.3.1	Implementation Aspects	156
6.3.2	Results and Discussion	157
7	Conclusion	159
	References	160
	List of Publications	177
	Author Biography	180

Chapter 1 Introduction

1.1 Theoretical Background on Secure Multi-Party Computation

Modern cryptography has been studied as a methods for secure communication in the environment with third parties called adversaries. In particular, since the standardization of the symmetric key cryptosystem DES in the late 1970s [103] and the proposal of public key cryptosystem by Diffie and Hellman [47], researches on cryptography have been rapidly developed and many research fields have been explored. One of the most important primitives in cryptography is an encryption, which is a technology for securely transmitting data to the receiver so as not to leak contents of the data to the eavesdropper (adversary). On the other hand, cryptography in a broad sense covers wider technologies besides encryption, such as key distribution, authentication, digital signature, pseudorandom number generation. In addition, one of the long-standing interests in cryptography is how/what kind of advanced functionalities can be achieved from these cryptographic primitives.

One of the oldest protocols that realizes such advanced applications is the “mental poker” protocol [113] by Shamir, Rivest and Adleman, which is a way to realize poker over the phone (without trusted third entity). In the same period, several cryptographic protocols were proposed, such as Yao’s millionaires’ problem [125] (how to know which of two persons is richer) and Even et al.’s document exchange protocol [50] (a method of exchanging documents simultaneously via telephone).

Later, in 1986, Yao generalized the above problems for the two party case [126]. Micali, Goldreich and Wigderson [60] also extended this to the n party case. These are the beginning of the research field known as *Multi-Party Computation (MPC)* in modern cryptography.

Examples of MPCs MPC protocols involves various applications of wide range. We introduce several examples of these protocols in the following.

- **Electronic auction** [52, 89, 95, 63, 19, 23]: The parties are auctioneers and

bidders. The auctioneer offers his/her goods, service or some contract. The bidders submit their own bids, and then the winner (e.g., who bids the highest price in the case of goods, or lowest price in the case of contract) gets the offered things. They want to perform auction without revealing each bidder's bid. The auctioneers and bidders finally know only the winner and his/her bids.

- **Electronic voting** [32, 15, 22, 55, 105, 96, 106]: The parties are voters and authorities managing the vote system. The voters want to confirm the result of auction without revealing each voters vote. In addition, the authorities want to ensure that the result of votes is correct (i.e., all vote is not tampered/substituted).
- **Database queries** [76, 54, 9, 2, 120]: The parties are client(s) and server(s). The client wants to ask queries against a database without revealing what he/she has been asked. The server wants to respond the queries without revealing contents of the database beyond the response for the queries. The queries involves keyword search, sorting, aggregation and so on.
- **Threshold cryptosystem** [21, 38, 45, 46, 58, 114, 115, 90, 86]: The parties are multiple users (typically signers or receivers). The threshold cryptosystems is a kind of cryptosystems that multiple parties cooperatively perform cryptographic operation such as signing or decryption. Such schemes are constructed so that if all (or a number more than threshold) of parties join the protocol, then signing/decryption procedure is succeeded. The purpose of threshold cryptosystems involves decentralizing authorities, or key escrow without revealing certificate or secret key against any user.

Since the research area of MPC is extraordinarily wide, it is difficult to talk about all of the MPC applications here. However, basically, MPC could be a good solution if we want to process any private data on a distributed environment. In recent days, it is considered that one of the most attracting applications of MPCs is so-called “privacy-preserving data analysis (mining)”. We will describe this topic in more detail in Section 1.2.

Models of MPCs We briefly describe a model of MPCs in the following.

Now we consider a network composed of n participants (a participant is called as “party”). Suppose that each party P_i ($i \in \{0, \dots, n-1\}$) holds his/her own secret x_i . We consider the following problem: When a function (also called functionality) f (for example, the sum or the maximum value) is given, each party wants to obtain $y = f(x_0, \dots, x_{n-1})$ (e.g. $y = x_0 + \dots + x_{n-1}$, $y = \max\{x_0, \dots, x_{n-1}\}$) while keeping their own information secret. The parties try to realize a functionality such that all of the parties finally know the value of y correctly, by the distributed computing among n parties. In particular, during the computation, each party can communicate with other parties. It is also required that the communication must not reveal their secret. The protocol that realizes such a functionality is called *a MPC protocol for the functionality f* .

Assuming a trusted third party (TTP) and the secure channel, this functionality is trivially realized by the following way: each party P_i sends x_i to TTP using the secure channel, and TTP calculates y and then send it back to each P_i . However, the assumption of a reliable TTP is strong and depends on non-technical factors (operation and maintenance in TTP, credit examination for TTP, etc.). Therefore, generally in MPC, only technically reasonable assumptions such as existence of secure channel (presence of symmetric key cryptosystem) and existence of public key cryptosystem are considered.

We also assume that adversaries are included among the n participants. We can consider two types of attacks by the adversaries: (1) to steal other parties’ secret inputs or (2) to tamper the output of MPC protocol illegally. For achieving both attacks, the adversaries can arbitrarily collude to achieve their goals. If an MPC protocol does not allow to steal or to tamper the output by t parties out of n , the MPC protocol is said to be t -secure. This security guarantees are also to be considered how adversaries behave in the protocol. There are two typical adversary models: *semi-honest* adversaries and *malicious* adversaries. (1) semi-honest adversaries: the adversaries follow the protocol specification but may try to learn more than allowed from the protocol transcript. (2) malicious adversaries: the adversaries can run any arbitrary polynomial-time attack strategy (i.e., adversary can deviate from the protocol specification).

On the type of assumptions for guaranteeing security, two models can be considered: *information-theoretic* model and *computational* model. In the information-theoretic model, security is obtained unconditionally and even in the presence of computationally unbounded adversaries (“information-theoretically secure” is also said as “unconditionally secure”).

More precisely, we say that a scheme is *perfectly secure* if any information the adversaries can obtain by the execution of the scheme (e.g., ciphertexts, transcripts) does not increase the success probability of an attack. On the other hand, we say that a scheme is *statistically secure* if the information that can be obtained from the scheme does not increase success probability of an attack *except tiny probability*.

More precisely, if the probability that the adversaries succeed the attack is exactly 0, the security is called as *perfect security*, and else if we allow the adversaries to succeed with a negligible probability, the security is called as *statistical security*. In contrast, in the computational model, security is obtained in the presence of polynomial-time adversaries and relies on computational hardness assumptions.

Research Direction First construction of secure 2-party computation was proposed in Yao’s paper in 1986 [126]. After that, first MPC protocol for n -party case and general functionality was shown [60], which depends on Yao’s idea, zero-knowledge proof [61] and verifiable SS [35]. In [60], the following result was shown: *assuming the existence of public key encryption (one-way trapdoor function), there is a t -secure multi-party protocol for any functionality¹ f and $t < n/2$.*

With these results as a start point, a large number of MPC protocols have been explored. Since the feasibility on MPC protocols for any functionality is already shown in the above results, the main interest on these research was mainly concentrated on improvement in terms of assumptions, threshold of the adversaries and efficiency. Generally it is required to design appropriate MPC protocols according to the purpose in each context, while considering these features.

- **On assumptions — From what kind of assumptions can we construct MPC protocols?:** One interest topic on assumptions is to characterize necessary/sufficient assumptions required to realize a functionality on MPC. In other words, we want to construct MPC protocols from weaker assumption as

¹When we argue about “any function”, we regard f as a Boolean circuit. Note that a circuit can approximate arbitrary continuous function with arbitrary precision

much as possible. For example, [61] assumes the existence of public key encryption. We can consider much weaker assumption like the existence of secure channel, or the existence of broadcast channel [36] (a channel for transmitting the same information to all parties simultaneously). For instance, the existence of secure channel is a weaker assumption than the existence of public key cryptography. Note that the existence on public key encryption is well known to depend on computational assumption (existence of a one-way trap-door function), whereas the existence of secure channel requires for constructing information-theoretically secure protocols.

- **On the threshold — How can we obtain MPC protocols with larger t ?:** As mentioned above, the security for MPC protocols with n parties are characterized by the threshold t which implies the maximum number of adversaries, and the first result [60] shows a feasibility on $t < n/2$ (namely, the case where honest parties are majority). It is considered the threshold t should be as large as possible. In addition, upper/lower bounds of t are also a theoretically interesting topic.
- **On efficiency — Can we construct more efficient MPC protocols?:** Theoretically, the efficiency of MPC protocols is evaluated by two metrics: *communication complexity* and *round complexity*. The communication complexity means the total amount of communication bits during the execution of MPC protocols. The round complexity means that the number of communication (interaction) during the MPC protocols.

In 1988, Ben-Or et al. [14] and Chaum et al. [33] showed the following result: “*assuming secure channel, for any functionality f and for $t < n/3$, there exist t -secure MPC protocols for f* ”. After that, in 1989, Rabin and Ben-Or [109] improved the upper bound of t to $t < n/2$. More precisely, they showed that “*assuming the secure channel and broadcast channel, for any functionality f and $t < n/2$, there exists t -secure MPC protocols for f* ”.

The above results [14, 33, 109] hold under on information-theoretic assumption. It seems that achieving larger upper bound of t than $t \geq n/2$ is difficult. Therefore, there are researches that aim to improve upper bound of t under computational

assumptions. For example, Goldreich, Micali and Wigderson [60], Beaver and Goldwasser [12] show that: “*assuming the existence of an oblivious transfer [108], for any functionality f and $t < n$, there exists t -secure MPC protocols for f* ” (with negligible error).

We summarize these known feasibility results in Table 1.1.

Table 1.1: Known Feasibilities on MPC

Adversaries	Model	Bound on t	Assumptions
Semi-honest	Information Theoretic (perfect security)	$t < \frac{n}{2}$ (Ben-Or et al. '87, Chaum et al. '87)	Secure channel
	Computational	$t < n$ (Goldreich et al. '87)	Secure channel, Oblivious Transfer
Malicious	Information Theoretic (perfect security)	$t < \frac{n}{3}$ (Ben-Or et al. '87, Chaum et al. '87)	Secure channel
	Information Theoretic (statistical security)	$t < \frac{n}{2}$ (Rabin, Ben-Or, '87)	Secure channel, Broadcast channel
	Computational	$t < n$ (Beaver, Goldwasser, '89)	Oblivious Transfer

On the efficiency, the MPC protocols described above all requires polynomial order of round complexity. Namely, when we want to compute large circuit on these MPC, it requires larger number of rounds. However, in 1990, Beaver et al [11] show that there exists MPC protocol whose round is constant (independent of f). In addition, it is also shown that there exists more dedicated construction for MPC protocols for several special functionality (comparison and so on) [39, 99, 119, 97, 98] (although these constant-round MPC protocols is asymptotically efficient but still have drawback that communication complexity in reasonable parameter is very large).

Well-known framework for constructing MPC protocols Here we introduce well-known frameworks for constructing MPC protocol for any functionality f (general-purpose MPC). In this thesis, “MPC protocol” means general-purpose MPC unless otherwise noted.

- **How can we obtain MPC protocols for “any functionality”?:** Since a functionality f can be considered as a circuit, we consider how to construct

MPC protocol for any circuit. Here we introduce a concept of “*functional completeness*” [49] in the theory of logical operations. It is known that there exists a specific set of logical operators (gates) can represent an arbitrary circuit². If a set of logical gate can represent circuits for arbitrary f , we call the set of logical gate is *functionally complete*.

Namely, if we can construct MPC protocols for a certain functionally complete set of logical gates, we can obtain a MPC protocol for any functionality f . An example of functionally complete set of logical gate is $\{XOR, AND\}$, where XOR and AND are binary operators over $\{0, 1\}$ (namely, \mathbb{Z}_2). In addition, instead of $\{XOR, AND\}$, we can consider $\{+, \cdot\}$ where “+” and “ \cdot ” are operator for addition and multiplication over \mathbb{Z}_q where $q \in \mathbb{N}$ ($q \geq 2$). Note that if $x, y \in \{0, 1\}$ then $x XOR y = (x + y) - (x \cdot y)$ and $x AND y = (x \cdot y)$. It means that $\{+, \cdot\}$ can be reduced to $\{XOR, AND\}$, and therefore $\{+, \cdot\}$ also can construct any functionality.

Particularly important methods to realize MPC protocols are the “*garbled circuit (GC)*” introduced by Yao [126] and MPC based on the *secret sharing (SS)* [112, 16] introduced by Ben-Or et al. [14] and Goldreich et al. [60].

GC-based: Garbled circuit is an method of 2-party computation for general Boolean circuit. In GC, the functionality f to be computed is represented as the composition of XOR gate and AND gate. For each gate, one of the parties (called “garbler”) make encrypted truth table by symmetric key encryption and send it to other party (called “evaluator”). Then, garbler also send the secret keys for encrypting truth table that corresponding to each parties input value. This procedure requires oblivious transfer [108], which is a cryptographic protocol in which the sender can not know which of the data sent by the sender has been received by the receiver.

SS-based: SS is one of the most important building blocks to construct MPC protocols independently introduced by Shamir [112] and Blakley [16]. SS divide the secret into n pieces (called “shares”), and each share is held by n parties.

²“arbitrary circuit” here means that arbitrary truth table for n -input and m -output function $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$ for any $n, m \in \mathbb{N}$

This shares has the properties that (1) the original secret information can be restored only when k shares among the n shares are collected, and (2) any information about the original secret is leaked from any $k - 1$ shares. In particular, if the reconstruction of the secret from the shares is a linear mapping, the SS is called as linear secret sharing (LSS). For constructing MPC based on SS, a LSS is often deployed.

Now, suppose that secret information a and b are respectively distributed to n parties by a LSS. At this time, the share of $a + b$ can be obtained only by adding up the shares held by each participant due to the linearity of LSS. In other words, we can easily construct a MPC protocol for “addition”, which computes the share of $a + b$ from the share of a and the share of b . Therefore if we can construct MPC for multiplication, we can compute arbitrary functionalities on MPC by combination of addition and multiplication. Although many SS-based MPC have been proposed until today, almost of known schemes are consist of “MPC for addition by LSS + individually designed MPC for multiplication”.

In recent days, MPC protocols can not be classified simply into these two styles, since several MPC protocols are consist of both of GC and SS.

The standard GC is for 2-party as mentioned above. However, Beaver et al. [13] introduced that how to extend the GC into n -party case by combining SS (this technique is called “BMR protocol” from the initial letters of the authors). Even in today, it is the well-used baseline for efficient constant round MPC protocol [88, 64].

ABY framework [44, 92] is a mixed-protocol of SS-based and GC-based MPC. The aim of this scheme is efficiently combining arithmetic operation and Boolean operation to utilize each own advantage of GC and SS.

On the efficiency of these techniques Comparing the above two methods, the advantage of GC is the small round complexity: it takes small constant round for any Boolean circuit MPC. On the other hand, GC requires encrypting truth tables for each gate f by symmetric key encryption, and it involves a much amount of communication than SS-based MPCs (since even for 1-bit information, a ciphertext of symmetric key encryption takes κ bits where κ is a security parameter). By recent progress [127], the number of ciphertext for each gate is reduced two, whereas the original Yao’s GC requires four ciphertexts for each gate. However, [127] also proved

that “two ciphertexts per gate” is lower bound in the standard garbling method.

SS-based MPC has opposite features to GC. Namely, it requires less amount of communication bits but takes large number of communication round. SS-based MPCs are basically requires communication for every multiplication. Therefore, its communication round depends on the “depth” of multiplication in the functionality f . However, the amount of computation once is very small. This is because the SS is based on information theoretic security, so it does not require the amount of communication dependent on security parameters like size of ciphertext.

1.2 Practical Background on Secure Multi-Party Computation

So far we have introduced the theoretical background of MPC. In the following, we will touch on the history of research on applications and implementations of MPC.

The root of applied research on MPC is “Privacy Preserving Data Mining” [87] by Lindell and Pinkas (Agrawal has also published a paper of the same name in another field in the same year [3]). In this paper, ID3 (Iterative Dichotomiser 3) algorithm, which is a kind of machine learning algorithm, is constructed using MPC protocol. Noteworthy, in this paper, a dedicated protocol is proposed to efficiently execute some processing such as logarithmic function required to realize ID3 in MPC. This paper is an important since it points out that the effectiveness of dedicated design for practical application, as opposed to the common knowledge of that “MPC can theoretically execute arbitrary processing”.

In 2004, the first implementation of MPC named “FairPlay” was proposed by Malkhi et al. [91]. Their implementation was based on Yao’s Garbled Circuit, and had no dedicated functionalities for advanced processing. The performance was 13ms per one logic gate (one billion times slower compared with a normal PC at that time), but it was the result that the efficiency of general purpose MPC was shown for the first time.

As an application to actual data, an experiment on application to a sugar beet auction by Bogetoft et al. was conducted in 2008 [19]. The adopted protocol is a scheme based on Shamir’s SS, and calculations mainly consist of addition, multiplication and comparison were performed in about 30 minutes. This work has been taken over by an auction solution provided by Partisia, Denmark.

Beginning with the above, many application of MPC are being searched. Among them, the recent trend of data analysis based on machine learning can not be ignored. Machine learning has a much deeper history than modern cryptography. However, the recent theoretical improvements and computer performance improvements allows developing remarkable application of deep neural networks (a.k.a deep learning). These achievements triggered the public to have hope for “artificial intelligence (AI)”.

With the focus on data analysis using machine learning, the privacy of training data has been raised as a problem. For example, according to McKinsey’s reports [66, 65], it is said that US healthcare community is able to generate more than \$300 billion in value each year if hospitals, caregivers and pharmaceutical companies can share and utilize data. However, such data sharing between different industries is difficult from the viewpoint of individual privacy.

Theoretically, MPC can compute arbitrary functionalities among many people while keeping the input secret. In other words, it is also possible to execute the machine learning algorithm while concealing the training data. In addition, the security of MPC protocols is supported by the well-developed rich cryptographic theory. Therefore, MPC has attracted the attention as the method for solving privacy issue on machine learning, and improving efficiency of MPC protocols is an important research theme, not only academically but also industrially.

What kind of “efficiency” is important in practical sense? From practical point of view, the efficiency of implementation for MPC protocols are measured by two metrics: *latency* and *throughput*. The latency means that the time from starting MPC protocol to ending the protocol (usually including the sharing input and reconstructing output). The throughput means that the number of MPC execution can be performed in a certain period (e.g., per second). Generally, if the latency is improved the throughput is also improved. The throughput is meaningful when we consider processing a large number of data utilizing parallelization. For example, if the MPC against single data can be done in 100ms latency, the throughput of this MPC is 10 processes/sec naively. However, if we can execute the MPC against 1,000 (independent) data in parallel, the throughput of this MPC can be 10,000 processes/sec.

In use cases of data analysis (such as data mining), throughput is more important than latency, since the scene requiring real-time processing is not so much. For example, in some cases it is sufficient to take execution time from night to the next

morning. In other cases, it is allowed that an analysis may take a week or a month.

Which MPC is better? On the latency, the GC and SS-based MPCs reaches same level. Typically, in the researches on MPC implementation, MPC for AES (a kind of symmetric key encryption) circuit are often used as benchmarking latency and throughput of the MPC protocol. According to [80], the SS-based MPC takes 14.3ms latency for one execution of AES circuit with semi-honest adversaries. On the other hand, [62] reported that GC takes 16ms with semi-honest adversaries.

For aiming high throughput in these method, the biggest problem is the communication complexity and physical limitation of communication channel.

As mentioned above, the GC requires larger amount of communication than SS-based scheme. Here we briefly estimate the communication bits that GC requires. Assuming 128-bit security and deploying AES for symmetric key encryption, one ciphertext of AES is 128 bits. Therefore, if we apply the method of [127], the GC requires $128 \times 2 = 256$ bits for each AND gate in the functionality f to be computed. We additionally let the two parties are connected by 10Gbps network. In this case, the upper bound of the throughput depending on communication bandwidth is roughly 7,500 AES/sec. In contrast, [18] reported that their SS-based MPC perform roughly 90,000 AES/sec.

The above estimation shows us the important facts: *GCs are not suitable for the purpose of obtaining a high throughput.* Due to the feature of MPC that communication is necessary, the efficiency of MPC is bounded by physical ability of communication channel. Therefore, in particular, the throughput of MPC with large amount of communication is very limited. In addition, [127] shows that the number of ciphertexts in GC is already optimal. Namely, it seems to be hard to improve the throughput of GC-based MPC.

Therefore, to obtain high throughput MPC, at least we should take SS-based approach. In this thesis, we follow SS-based approach to get high throughput MPCs.

1.3 Motivation and Our Results

From the above observation, a design of MPCs optimized for small communication and parallelization is considered to be important to construct high throughput MPCs.

In this thesis, we aim to provide efficient MPC protocols design and implementation achieving high-throughput. This challenge in achieving this is both on the

computational and network levels. More precisely:

- We aim to design a high-throughput MPC. We deploy SS-based MPC in order to reduce communication complexity, and basically do not deploy computationally secure primitive for construction (except for implementing pseudo random generator, since the information-theoretic random number generation requires much amount of communication).
- To explore the limits of efficient MPC, we particularly focus on 3-party and honest-majority setting as a first step. The number of parties affect the communication cost of MPC protocols since they should communicate with each other. In addition, 3-party is the minimal case in which feasibilities on information-theoretic assumption is shown. Despite of its advantage, the known latest result on this setting [18] still does not seem to meet the performance requirements of realistic tasks.
- To eliminate gaps between theory and implementation, we also aim to provide optimized implementation design. In particular, we basically have been focused on the overhead of communication. However, in the real world, MPC protocols also require local computation of each parties, and it might be an obstacle for the MPC protocols to make the best of the performance of the communication environment. Therefore, we also focus on fast implementations of MPC to minimize computation cost. It involves optimizations utilizing the cache memory and CPU instruction sets for vectorization.

The result of this papers are four-folded as follows. The results 1. and 2. are *foundations* of this thesis, which involve efficient constructions for baseline MPC protocols (for gate level) with semi-honest/malicious adversaries and its optimized implementations. On the other hand, the results 3. and 4. are *applications*, which involve methods for efficiently constructing MPC protocols for arbitrary (more complex than gates) functionalities from the MPC in the result 1. and 2.

1. **Foundation (1): High-throughput semi-honest secure 3-party computation based on replicated SS with honest-majority:** In this thesis, we describe a new information-theoretic protocol (and a computationally-secure

variant) for secure three-party computation with an honest majority. The protocol has very minimal computation and communication; for Boolean circuits, each party sends only a single bit for every AND gate (and nothing is sent for XOR gates). Our protocol is (simulation-based) secure in the presence of semi-honest adversaries, and achieves privacy in the client/server model in the presence of malicious adversaries.

We demonstrate the practical potential of our protocol by implementing a MPC-based system for Kerberos authentication, which is a well-known network authentication protocol based on symmetric-key cryptosystems. Our MPC-Kerberos system can support a login storm of over 40,000 user authentications per second, which is sufficient even for very large organizations.

2. **Foundation (2): Optimizing cheating detection for honest-majority**

MPC: We provide general techniques for improving efficiency of cut-and-choose protocols on multiplication triples and utilize them to significantly improve the recently published protocol of Furukawa et al. [56]. We reduce the bandwidth of their protocol down from 10 bits per AND gate to 7 bits per AND gate, and show how to improve some computationally expensive parts of their protocol. Most notably, we design cache-efficient shuffling techniques for implementing cut-and-choose without randomly permuting large arrays (which is very slow due to continual cache misses). We provide a combinatorial analysis of our techniques, bounding the cheating probability of the adversary. Our implementation achieves a rate of approximately 1.15 billion AND gates per second on a cluster of three 20-core machines with a 10Gbps network. Thus, we can securely compute 212,000 AES encryptions per second (which is hundreds of times faster than previous work for this setting). Our results demonstrate that high-throughput secure computation for malicious adversaries is possible.

3. **Application (1): Compiler for SS-based MPCs:**

Today, we have protocols that can carry out large and complex computations in very reasonable time (and can even be very fast, depending on the computation and the setting). Despite this amazing progress, there is still a major obstacle to the adoption and use of MPC due to the huge expertise needed to design a specific MPC

execution. In particular, the functionality to be computed needs to be represented as an appropriate Boolean or arithmetic circuit, and this requires very specific expertise. In order to overcome this, there has been considerable work on compilation of code to (typically) Boolean circuits.

In this thesis, we design and implement a MPC compiler for our three-party honest majority MPC. Our implementation is an extension of a well-known MPC compiler called “SPDZ compiler” so that it can work with general underlying protocols. In this thesis we called the compiler we made “generalized SPDZ compiler”. Moreover, our SPDZ extensions were made in mind to enable the use of SPDZ for arbitrary protocols and to make it easy for others to integrate existing and new protocols.

We integrated three different types of protocols: (1) an honest-majority protocol for computing arithmetic circuits over a field (for any number of parties), (2) a three-party honest majority protocol for computing arithmetic circuits over the ring of integers \mathbb{Z}_{2^n} , and (3) the multi-party BMR protocol for computing Boolean circuits. We show that a single high-level SPDZ-Python program can be executed using all of these underlying protocols (as well as the original SPDZ protocol), thereby making SPDZ a true general run-time MPC environment.

4. **Application (2): Dedicated MPC protocols for high-level functionalities** Although our SS-based 3-party MPC proposed in the above results is very efficient in general, the SS-based MPCs are still inefficient for several heavy computations like algebraic operations, as they require a large amount and number of communication proportional to the number of multiplications in the operations (which is not the case with other SS-based MPCs). In this thesis, we propose the following two dedicated MPC protocols for high-level functionalities to accelerate SS-based MPC further.

Arithmetic-to-Boolean/Boolean-to-Arithmetic Conversion Most real-world programs consist of a combination of arithmetic and non-arithmetic computations, and thus need a mix of arithmetic and Boolean low-level operations. In order to facilitate this, we propose new MPC protocols named *bit decomposition* and *ring composition* operations, to convert a shared ring element to a series of shares of its bit representation and back.

Compared with the previous best protocols, our bit decomposition and ring composition achieve two order of magnitude less communication bits in 32-bit integer case, which is considered as a reasonable parameter. The protocols are integrated into the generalized SPDZ compiler described above and thus we can see the practical efficiency of these protocols in the complex mixed operation of arithmetic and Boolean, like SQL query on fixed-point numbers.

Modular Exponentiation As one of the most popular algebraic operations, we propose RSSS-based three party computation protocols for modular exponentiation on the case where the base is public and the exponent is private. We will show the practical effect of our protocol by experiments on the scenario for distributed signatures, which is useful for secure key management on the distributed environment (e.g., distributed ledgers). Our protocols are more efficient in terms of both of communication complexity and round complexity than previous standard scheme. More precisely, for the size of secret values n , the proposed schemes require $O(n)$ bits communication whereas the previous scheme requires $O(n^2)$ bits. As for the round complexity, a several variants in our proposal require $O(n)$ round as same as previous scheme, and other variants in our proposal require just $O(1)$ rounds.

1.4 Organization of the Thesis

The organization of this thesis is as follows.

Chapter 2 shows a common notation and definition used in the later chapters. Each chapter after Chapter 3 corresponds to one topic of the result shown in Section 1.3. In Chapter 3 (corresponds the result 1.) we describes a new protocol for the SS-based MPC protocol with 3-party, semi-honest adversaries and honest-majority settings. We also show its optimized implementation and experimental results. Chapter 4 (corresponds to the result 2.) shows our method for improving cheater detection protocol, and how to apply it to the semi-honest MPC described in Chapter 3 to obtain maliciously-secure 3-party MPC. In addition, we also show the optimized implementation the experimental results. Chapter 5 (corresponds to the result 3.) contains our design of MPC compiler for our 3-party MPCs based on SPDZ

framework. We also show experimental results on comparison between other state-of-the-art MPCs which work on SPDZ-based compiler, and see the effectiveness of our 3-party MPC protocol. Then, in Chapter 6 (corresponds to the result 4.), we describe our new MPC protocols for arithmetic-to-Boolean/Boolean-to-Arithmetic conversions and modular exponentiation based on SS-based MPCs in Chapter 3 and 4. We also show the experimental results with semi-honest adversaries and see the effectiveness of the protocol. Finally, we conclude this thesis in Chapter 7.

Chapter 2 Preliminaries

2.1 Notation

- Let \mathbb{Z} be a set of integer, \mathbb{N} be a set of the natural number (including 0) and \mathbb{R} be a set of the real number.
- Let $\mathbb{Z}_q = \{0, 1, \dots, q-1\}$ be a quotient ring with positive integer q . Note that if $q = p$ where p is prime \mathbb{Z}_p is a *field*.
- Let $\{0, 1\}^k$ be a set of bit strings with length k , and $\{0, 1\}^* = \cup_{k \in \mathbb{N}} \{0, 1\}^k$ (a set of bit strings with finite length).
- Let $[0, 1]$ be a closed interval, which is a set of real numbers such that any $x \in [0, 1]$ satisfies $0 \leq x \leq 1$.
- A probability space is a triple $(\Omega, \mathcal{F}, \Pr)$, where Ω is called a sample space, $\mathcal{F} \subseteq 2^\Omega$ is called an event space and $\Pr : \mathcal{F} \rightarrow \mathbb{R}$ calls a probability measure function such that:
 - \mathcal{F} is modeled as σ -algebra on Ω , which satisfies the following properties (1)–(3): (1) $\Omega \in \mathcal{F}$, (2) if $E \in \mathcal{F}$, then $\Omega \setminus E \in \mathcal{F}$, (3) if $E_i \in \mathcal{F}$ for all $i \in \{1, \dots, \infty\}$, then $\cup_{i=1}^\infty E_i \in \mathcal{F}$.
 - \Pr satisfies following properties (a)–(c): (a) for any $E \in \mathcal{F}$, $0 \leq \Pr[E] \leq 1$, (b) $\Pr[\Omega] = 1$, (c) for any countably infinite sequence of pairwise disjoint events E_1, E_2, E_3, \dots , $\Pr[\cup_{i=1}^\infty E_i] = \sum_{i=1}^\infty \Pr[E_i]$.

We call $\Pr[E]$ the probability of the event $E \in \mathcal{F}$.

- A (discrete) random variable X on a sample space Ω is a function on Ω that takes value on finite or countably infinite number of values. Namely $X : \Omega \rightarrow \mathcal{E}$. For all $x \in \mathcal{E}$ we use the notation “ $X = x$ ” to denote the event $X^{-1}(x) = \{s \in \Omega \mid X(s) = x\}$, which means all the basic events of the sample space in which the random variable X assumes the value x (namely, $\Pr[X = x] = \Pr[X^{-1}(x)]$).

The probability distribution of X is defined by $p_X : \mathcal{E} \rightarrow [0, 1]$ of X such that $p_X(x) = \Pr[X = x]$.

- For a random variable $X : \Omega \rightarrow \mathcal{E}$, “ $x \leftarrow X$ ” denotes that x is sampled from \mathcal{E} according to the probability distribution of X .
- Let \mathcal{P} be a set of parties and $P_i \in \mathcal{P}$ be a party with the identifier i . In this paper we use zero-based numbering for the parties and corresponding shares. Namely, the indices are started from the number 0.

2.2 Indistinguishability

Here we introduce a definition for the statistical distance and indistinguishability for defining security in later sections. We note that there are three types of definitions for indistinguishability: perfect indistinguishability, statistical indistinguishability, computational indistinguishability.

Definition 2.2.1 (Perfect Indistinguishability). *Let $A = \{A_i\}_{i \in \{0,1\}^*}$ and $B = \{B_i\}_{i \in \{0,1\}^*}$ be probability ensembles. Let κ be a security parameter. We say that A and B are perfectly indistinguishable, denoted by $A \equiv B$, if*

$$\delta(A_i, B_i) = 0$$

for every $i \in \{0, 1\}^*$.

Definition 2.2.2 (Statistical Distance). *Let X and Y be two random variables over sample space Ω . The statistical distance between X and Y is defined by*

$$\delta(X, Y) = \frac{1}{2} \sum_{w \in \Omega} |\Pr[X = w] - \Pr[Y = w]|.$$

Definition 2.2.3 (Statistical Indistinguishability). *Let $A = \{A_i\}_{i \in \{0,1\}^*}$ and $B = \{B_i\}_{i \in \{0,1\}^*}$ be probability ensembles. Let κ be a security parameter. We say that A and B are statistically indistinguishable, denoted by $A \stackrel{s}{\equiv} B$, if for every non-uniform (computationally unbounded) algorithm \mathcal{D} there exists a function $p(\cdot)$ such that for every $i \in \{0, 1\}^*$ and every $\kappa \in \mathbb{N}$,*

$$\delta(A_i, B_i) \leq \frac{1}{p(\kappa)}.$$

Definition 2.2.4 (Computational Indistinguishability). Let $A = \{A_i\}_{i \in \{0,1\}^*}$ and $B = \{B_i\}_{i \in \{0,1\}^*}$ be probability ensembles. Let κ be a security parameter. We say that A and B are computationally indistinguishable, denoted by $A \stackrel{c}{\equiv} B$, if for every non-uniform polynomial-time algorithm \mathcal{D} there exists a function $p(\cdot)$ such that for every $i \in \{0,1\}^*$ and every $\kappa \in \mathbb{N}$,

$$|\Pr[\mathcal{D}(A_i) = 1] - \Pr[\mathcal{D}(B_i) = 1]| \leq \frac{1}{p(\kappa)}.$$

2.3 Secret Sharing

Here we describe the formal definition of secret sharing (SS), which is an important building block to construct MPC protocols. SS were proposed by Shamir [112] and Blakley [16] independently.

In SS, secret information is divided into multiple data called “shares”. This share is created so that the original secret information can be reconstructed only when certain combinations are collected. The most popular SS scheme used in MPC is the (k, n) -threshold scheme. It has the property that the original information can be reconstructed by collecting any $k (\leq n)$ shares among the n shares, and that no information on the secret can be leaked from any $k - 1$ shares.

2.3.1 (k, n) -threshold schemes

Definition 2.3.1 ((k, n) -threshold scheme). A (k, n) -threshold scheme is a set of the following two probabilistic algorithms **Share** and **Reconst** with finite space \mathcal{M} and \mathcal{S} such that:

- **Share**: Given a secret $m \in \mathcal{M}$, the algorithm **Share** outputs n shares $\vec{s} = ([m]_0, \dots, [m]_{n-1}) \in \mathcal{S}^n$.
- **Reconst**: Given a k -tuple of shares, the algorithm **Reconst** outputs a message $m \in \mathcal{M}$.

and satisfying following two requirements.

- **Correctness**: We say a (k, n) -threshold scheme satisfies correctness (or a (k, n) -threshold scheme is correct) if the following property is satisfied: $\forall m \in \mathcal{M}, \forall I = \{i_0, \dots, i_{k-1}\} \subseteq \{0, \dots, n-1\}$ of size k ,

$$\Pr_{\text{Share}(m) \rightarrow ([m]_0, \dots, [m]_{n-1})} [\text{Reconst}([m]_{i_0}, \dots, [m]_{i_{k-1}}) = m] = 1.$$

- **k -out-of- n Perfect Privacy:** Let M be a random variable that takes value on \mathcal{M} and S_i ($i \in \{0, \dots, n-1\}$) be a random variable that takes value on \mathcal{S} . We say a (k, n) -threshold scheme satisfies k -out-of- n perfect privacy (or a (k, n) -threshold scheme is perfectly private) if the following property is satisfied: $\forall m \in \mathcal{M}, \forall \mathcal{I} = \{i_0, \dots, i_{k-2}\} \subset \{0, \dots, n-1\}$ of size $k-1$ and $\forall [m]_{i_0}, \dots, [m]_{i_{k-2}} \in \mathcal{S}$,

$$\Pr[M = m] = \Pr[M = m \mid S_{i_0} = [m]_{i_0}, S_{i_1} = [m]_{i_1}, \dots, S_{i_{k-2}} = [m]_{i_{k-2}}]$$

By $[m]$ we denote that the secret m is shared among n parties P_0, \dots, P_{n-1} and P_i holds the share $[m]_i$. In addition, $[m]^q$ denotes that m is shared by SS over \mathbb{Z}_q (namely, $\mathcal{M} = \mathbb{Z}_q$ and $\mathcal{S} = (\mathbb{Z}_q)^d$ where $d \in \mathbb{N}$) and $[m]_i^q = ([m]_{i,0}^q, \dots, [m]_{i,d-1}^q)$ denotes the P_i 's share of $[m]^q$.

2.3.2 Replicated Secret Sharing

In this thesis, we employ an instantiation of (k, n) -threshold schemes called *replicated secret sharing*. The $(2, 3)$ SS of the replicated type described in [37]. Here we follow the replicated secret sharing which is used in Araki et al.'s scheme [6].

Definition 2.3.2. *Replicated $(2, 3)$ secret sharing is a set of the following two probabilistic algorithms **Share** and **Reconst**. We additionally let all indices corresponding the index space $\{0, 1, 2\}$ are described as over modulus 3 and hereafter we omit the description of “mod3”. For example, a certain value x_i indexed by $i \in \{0, 1, 2\}$, x_3 is handled as x_0 , and x_{-1} is handled as x_2 .*

Share: *Given a specification of \mathbb{Z}_q , an element of \mathbb{Z}_q $x \in \mathbb{Z}_q$, as (\mathbb{Z}_q, x) , the algorithm **Share** generates random elements $x_0, x_1, x_2 \in \mathbb{Z}_q$ under the condition of $x_0 + x_1 + x_2 = x$, generates a share of P_i denoted by $[x]_i^q$ as $(x_{i-1} + x_i, x_{i-1})$ for $i \in \{0, 1, 2\}$, and output a set of all shares $[x]^q$. Here, $([x]_{i,0}^q, [x]_{i,1}^q) = (x_{i-1} + x_i, x_{i-1})$.*

Reconst: *Given $(i, [x]_i^q, [x]_{i+1}^q)$ for $i \in \{0, 1, 2\}$, the algorithm **Reconst** outputs $x = [x]_{i,0}^q + [x]_{i+1,1}^q$.*

2.4 The Model of Secure Computation

In this section, we describe the model of MPC in this thesis.

2.4.1 Settings

In general, we can consider two kind of settings. One is the setting traditionally considered in the context of MPC: each party has own secret value and they perform and they will obtain values of some function on their private input by performing MPC. Another setting is called the “Client-Server” setting, which is introduced Sharemind [17]. In this model, the entities performing MPC protocol are “servers” and the entity(ies) to the role of “client(s)” serves input to servers and get output from these servers. Namely, the servers does not obtain the result of MPC itself. Due to this characteristic, its security guarantee is different from the general setting.

In the following, we formalize each of settings.

General Setting A multi-party protocol is specified by a (possibly probabilistic) procedure referred to as *functionality*. Denote $f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n$ as the functionality, where n is the number of inputs. In the general setting, we consider n parties: each party $P_i \in \mathcal{P}$ ($i \in \{0, \dots, n-1\}$) holds a secret value x_i and the parties want to agree on some functionality f that takes n inputs. Specifically, $f = (f_0, f_1, \dots, f_{n-1})$ and each party $P_i \in \mathcal{P}$ ($i \in \{0, \dots, n-1\}$) will obtain distinct outputs $f_i(x_0, \dots, x_{n-1})$ in general. In this thesis, we consider only the case where every party will obtain the same outputs (namely, $f_0 = f_1 = \dots = f_{n-1}$).

Client-Server Setting Here we assume there is t clients which has their private input x_i ($i \in \{0, \dots, t-1\}$), and they want to agree a t -input functionality $f = (f_0, \dots, f_{t-1})$ on their private inputs. The goal is that each client will obtain outputs $y_i = f_i(x_0, \dots, x_{t-1})$ and the servers will obtain nothing. As same as the general setting, we consider the case of $f_0 = f_1 = \dots = f_{t-1}$ in this thesis.

In both settings, we particularly consider 3-party computation (3PC). Namely, the parties are P_0, P_1 and P_2 .

2.4.2 Security Criteria

Basic Requirements

- **Correctness:** The parties P_0, \dots, P_{n-1} obtain correct output $f(x_0, \dots, x_{n-1})$ if the parties follow the protocol properly.
- **Privacy** Each party P_i cannot learn anything about the other party’s input from the information sent during the execution of MPC protocol. Namely, The

information that each party can obtain about the secret is whatever could be derived from the output $f(x_0, \dots, x_{n-1})$.

Semi-Honest vs. Malicious adversaries The security of secret sharing based secret computation guarantees that “If the number of malicious participants among n participants is less than a certain threshold number, the computation process will not leak information about the private inputs”. At this time, the behavior of malicious participants is roughly classified into two types.

- **Semi-Honest adversaries:** Attackers follow the protocol but do not tamper with the data. The purpose of this adversaries is to obtain the information to be concealed only from the information obtained by the normal execution of the protocol.
- **Malicious adversaries:** Attackers can actively deviate from the protocol to tamper any data during the execution of MPC. In this case, there are two possible purposes of the adversaries: (1) tampering the output of MPC, and (2) obtaining the input information by observing the output affected by the tampering.

We call the MPC protocol secure against semi-honest/malicious adversaries “semi-honest/maliciously secure MPC”, respectively.

Perfect Security vs. Computational Security Regarding the computational power of the adversaries, there are two types of security criteria: perfect security and computational security.

- **Perfect security:** When the security of a protocol can be proven against computationally unbounded adversaries, we call the protocol is perfectly secure. Perfect security is also known as unconditional security or information-theoretic security (since the security purely underlies on information theory). For example, the notion of k -out-of- n perfect privacy for secret sharing described in Sect. 2.3 is a kind of perfect security in terms of that less number of the shares than threshold leaks no information about the secret.
- **Computational security:** When the security of a protocol can be proven against computationally bounded adversaries, we call the protocol is computationally secure. More precisely, the computational power of the adversaries is

assumed to be of polynomial-time). Most of cryptographic tools, like encryption and signatures, relies on the computational security. Therefore, when we use cryptography in a protocol, the protocol is only expected to be secure against computationally-bounded adversaries because unconditional (unlimited) adversaries can break the security of computationally secure cryptographic tools.

In general, perfectly secure protocols requires large memory size but its computational complexity is very small compared with computationally secure protocols.

In the context of MPC, these characteristics has both advantages and disadvantages.

2.4.3 Simulation-based Security

The goal of MPC protocol based on secret sharing is to compute shares of outputs from shares of inputs without revealing information on the input.

The security of MPC is formalized by simulation-based security. Namely, if there exist simulators that can generate the view of each party in the execution from given inputs and outputs, the MPC protocol is secure. This formalization implies that the parties learn nothing about inputs from the execution of the protocol, except for the information derived from outputs.

We use the definition of security in the presence of semi-honest adversaries as in [28, 59], making the necessary changes to formalize perfect security as well.

Perfect security in the presence of semi-honest adversaries. Loosely speaking, a protocol is secure in the presence of one corrupted party if the view of the corrupted party in a real protocol execution can be generated by a simulator given only the corrupted party's input and output. The view of party i during an execution of a protocol π on inputs \vec{x} , denoted $\text{VIEW}_i^\pi(\vec{x})$, consists of its input x_i , its internal random number r_i and the messages that were received by i in the execution. The output of all parties from an execution of π is denoted by $\text{OUTPUT}^\pi(\vec{x})$.

The following is the security definition for 3-party functionalities.

Definition 2.4.1 (Perfect security for probabilistic 3-ary functionalities in the presence of semi-honest adversaries). *Let $f : (\{0, 1\}^*)^3 \rightarrow (\{0, 1\}^*)^3$ be a probabilistic 3-ary functionality and let π be a protocol. We say that π computes f with perfect security in the presence of one semi-honest corrupted party for f if there exists a probabilistic polynomial-time algorithm \mathcal{S} such that for every corrupted party $i \in \{0, 1, 2\}$,*

and every $\vec{x} \in (\{0, 1\}^*)^3$ where $|x_1| = |x_2| = |x_3|$:

$$\left\{ (\mathcal{S}(x_i, f_i(\vec{x})), f(\vec{x})) \right\} \equiv \left\{ (\text{VIEW}_i^\pi(\vec{x}), \text{OUTPUT}^\pi(\vec{x})) \right\} \quad (1)$$

If Eq. (1) holds with computational indistinguishability, then we say that π computes f with computational security in the presence of one semi-honest corrupted party.

The above definition is for the general case of probabilistic functionalities, where we consider the joint distribution of the output of \mathcal{S} and of the parties. For the case of deterministic functionalities, however, we can separate the correctness and privacy requirements, and use a simpler and easier to prove definition. As shown in [59] (see Section 7.3.1), *any probabilistic functionality* can be securely computed in the presence of t corrupted parties using a general protocol which computes *any deterministic functionality* in the presence of t corrupted parties. Therefore, in order to prove the security of our protocol we can use the definition for deterministic functionalities stated below.

Definition 2.4.2 (Perfect security for deterministic 3-ary functionalities in the presence of semi-honest adversaries). *Let $f : (\{0, 1\}^*)^3 \rightarrow (\{0, 1\}^*)^3$ be a deterministic 3-ary functionality and let π be a protocol. We say that π computes f with perfect security in the presence of one semi-honest corrupted party for f , if for every $\vec{x} \in (\{0, 1\}^*)^3$ where $|x_1| = |x_2| = |x_3|$, it holds that $\text{OUTPUT}^\pi(\vec{x}) = f(\vec{x})$, and there exists a probabilistic polynomial-time algorithm \mathcal{S} such that for every corrupted party $i \in \{0, 1, 2\}$, and every $\vec{x} \in (\{0, 1\}^*)^3$ where $|x_1| = |x_2| = |x_3|$:*

$$\{\mathcal{S}(x_i, f_i(\vec{x}))\} \equiv \{\text{VIEW}_i^\pi(\vec{x})\}.$$

We prove the security of our protocols using the hybrid model, where parties run a protocol with real messages and also have access to a trusted party computing an other functionality g for them. The theorem in [29] referred as “modular sequential composition theorem” states that replacing the trusted party computing the functionality g with a real secure protocol results in the same output distribution. For the functionality is g , we say that the protocol works in the g -hybrid model and g is a *subfunctionality* in the hybrid model.

Perfect security in the presence of malicious adversary Let $\text{VIEW}_{\mathcal{A}, I, \pi}(\vec{v}, \kappa)$ denote the view of an adversary \mathcal{A} who controls parties $\{P_i\}_{i \in I}$ (with $I \subset [n]$) in a

real execution of the n -party protocol π , with inputs $\vec{v} = (v_1, \dots, v_N)$ and security parameter κ . We stress that in this setting, the vector of inputs \vec{v} is of length N and N may be much longer (or shorter) than the number of parties n running the protocol. This is because N refers to the number of inputs and so the number of clients, whereas n denotes the number of servers running the actual protocol. In addition, the servers do not receive input any of the values in \vec{v} but rather they each receive *secret shares* of the value.

Definition 2.4.3 (Computational security in the client-server model in the presence of malicious adversaries). *Let $f : (\{0, 1\}^*)^N \rightarrow (\{0, 1\}^*)^N$ be an N -party functionality and let π be an n -party protocol. We say that π t -securely computes f in the client-server model in the presence of malicious adversaries if the output that each party finally obtain is correct and if for every non-uniform probabilistic polynomial-time adversary \mathcal{A} , every $I \subset [n]$ with $|I| \leq t$, and every two series of length- N vectors $V_1 = \{\vec{v}_\kappa^1\}$, $V_2 = \{\vec{v}_\kappa^2\}$*

$$\{\text{VIEW}_{\mathcal{A}, I, \pi}(\vec{v}_\kappa^1, \kappa)\}_{\kappa \in \mathbb{N}} \stackrel{c}{=} \{\text{VIEW}_{\mathcal{A}, I, \pi}(\vec{v}_\kappa^2, \kappa)\}_{\kappa \in \mathbb{N}}$$

where for every $\kappa \in \mathbb{N}$, $\vec{v}_\kappa^1, \vec{v}_\kappa^2 \in (\{0, 1\}^*)^N$ and all elements of \vec{v}_κ^1 and \vec{v}_κ^2 are of the same length.

Loosely speaking, a protocol is private in the presence of one malicious corrupted party if the view of the corrupted party when the input is \vec{v}_κ^1 is computationally indistinguishable from its view when the input is \vec{v}_κ^2 . In order to rule out a trivial protocol where nothing is exchanged, we also require correctness, which means that when all parties are honest they obtain the correct output.

Universal composability. Protocols that are proven secure in the universal composability framework [29] have the property that they maintain their security when run in parallel and concurrently with other secure and insecure protocols. In [78, Theorem 1.5], it was shown that any protocol that is proven secure with a black-box non-rewinding simulator and also has the property that the inputs of all parties are fixed before the execution begins (called **input availability** or **start synchronization** in [78]), is also secure under universal composability. Since the input availability property holds for all of our protocols and subprotocols, it is sufficient to prove security in the classic stand-alone setting and automatically derive universal composability from [78]. We remark that this also enables us to call the protocol and subprotocols

that we use in parallel and concurrently (and not just sequentially), enabling us to achieve more efficient computation (e.g., by running many executions in parallel or by running each layer of a circuit in parallel).

2.4.4 Representation of Functionalities for Secret Sharing-based 3PC

For each MPC operation, each party receives the operation code representing a functionality and its input as shares. Note that the operations to be computed are public for every parties. Here we call the representation of a functionality as “opcodes”. The whole computation to be computed among 3 parties is represented by a sequence of such opcodes and its input shares. For each opcode given the parties, they invoke the function corresponding the opcodes. In the process of this function, the parties communicate with each other as necessary.

For example, we consider the case of MPC protocol for multiplication represented by the opcode **mult**. When the parties starts MPC protocol for multiplication with shared input x and y , each party P_i ($i \in \{0, 1, 2\}$) takes (**mult**, $[x]_i^q$, $[y]_i^q$) as inputs, and invoke corresponding function **mult**($[x]_i^q$, $[y]_i^q$) then get $[z]_i^q$ where $z = x \cdot y$ (see also definition in Sect. 3.3.1). Namely, when each party calls **mult**($[x]_i^q$, $[y]_i^q$), the process executing **mult** performs MPC protocol for multiplication with each own shares $[x]_i^q$, $[y]_i^q$ (while communicating with other parties’ process executing **mult**), and finally obtain the share of $[z]_i^q$ as a return value of **mult**. To simplify the notation, we describe the opcodes and their corresponding functions by the same name.

Part I

Foundations: Secure 3-Party Computation for General Circuits — Theory and Implementations for More Efficient Primitives

Chapter 3 Foundation (1): Semi-Honest Secure 3-Party Computation based on Replicated Secret Sharing

3.1 Introduction

A new protocol. We describe a new three-party protocol that is both extremely simple and has seemingly optimal bandwidth. Our protocol is suitable for arithmetic circuits over any field or over the ring modulo 2^n . Addition gates require local addition only, and multiplication gates require that each party send just a *single* field/ring element to one other party. In the Boolean case, this means that each party transmits a *single bit* only per AND gate.¹ Furthermore, the computation in our protocol is extraordinarily simple: in the case of Boolean circuits, each party carries out a single XOR operation per XOR gate, and 2 AND and 3 XOR operations per AND gate. Since all operations are merely XOR and AND, this also lends itself to parallelization on standard computers (in particular, XOR and AND over 128 bit registers can be carried out in the same time as for a single bit using Intel intrinsics).

Security. We prove that our protocol is secure in the presence of semi-honest adversaries with at most one corrupted party, under the standard simulation-based definitions. The basis of our protocol is *information theoretic* (and in fact perfectly secure). However, we save on communication by generating correlated randomness computationally, and therefore our overall protocol is computationally secure. (This combination enables us to achieve simple operations and save on additional bandwidth.) In addition to the above, we also consider a client/server model where any number of clients send shares of their inputs to 3 servers that carry out the computation for the clients and return the results to them (without learning anything). This

¹This is “seemingly” optimal in terms of bandwidth, but this has not been proven and seems hard to do so; see [67].

model makes sense for “outsources secure computation services” and indeed is the business model of Cybernetica. We show that in this model, our protocol actually achieves *privacy in the presence of malicious adversaries*, meaning that a single malicious server cannot learn anything about the input or output. (We stress that this notion is strictly weaker than simulation-based security in the presence of malicious adversaries, and in particular, does not guarantee correctness. Nevertheless, it does guarantee that privacy is not breached even if one of the servers behaves maliciously.)

Number of parties. As in Sharemind [17, 18], our protocol is specifically designed for 3 parties with at most one corrupted. This is unlike BGW [14] that works for any number of parties with an honest majority. An important open question left by this paper is the design of a protocol with comparable complexity that works for any number of parties. This seems to be very challenging, based on attempts that we have made to extend our protocol.

Experimental results. We implemented our new protocol for Boolean circuits in C++ and using standard optimizations. In order to take advantage of the very simple operations required in our protocol, we used Intel intrinsics in order to carry out many executions in parallel. This is described in detail in Section 3.6.1. We ran our experiments on a cluster of three nodes, each with two 10-core Intel Xeon (E5-2650 v3) processors and 128GB RAM, connected via a 10Gbps Ethernet. (We remark that little RAM was utilized and thus this is not a parameter of importance here.) We carried out two main experiments, both based on securely computing the AES circuit on shared keys.

First, we computed AES in counter mode, with the aim of obtaining maximal throughput. Using the full power of the cluster (all cores), we computed over 1.3 million AES operations per second. Furthermore, utilizing a single core we achieved 100,000 AES operations per second, and utilizing 10 cores we achieved almost 1 million AES operations per second. As we will show below in Section 3.2, this way outperforms all previous protocols of this type.

Second, we wished to demonstrate that this type of protocol can be incorporated into a real system. We chose to integrate our protocol into a Kerberos KDC in order to carry out Ticket-Granting-Ticket encryption without any single server holding the encryption key (whether it be a server’s key or user’s hashed password). Such an architecture protects against administrators stealing passwords, or an attacker who

breaches the network being able to steal all users' passwords. (We stress that in Kerberos, the raw password is never used so once the hashed password is stolen the attacker can impersonate the user.) We obtained a latency of 110ms on the server and 232ms on the client (over a LAN) for the *entire Kerberos login* (excluding database lookup). Given that this is for the purpose of user authentication, this is well within the acceptable range. In addition, we are able to support a login storm of over 40,000 user authentications per second, which is sufficient even for very large organizations.

Our results demonstrate that secure computation can be used to solve large-scale problems in practice (at least, for the cases that semi-honest security or privacy for a malicious adversary suffices).

3.2 Related Work

We compare our results with previously reported results on secure AES computation for 3 parties with an honest majority and semi-honest adversaries; see Table 3.1.

We stress that this table gives only very partial information since different hardware was used for each; we provide it to show the progress made and where we fit into it. However, fortunately, the setup used by us is *almost the same* as that of the latest Sharemind results in [110] (using optimized code that was completely rewritten), and we now provide an in-depth comparison to it. The benchmarking in [110] was carried out between three computers with two 8-core Intel Xeon (E5-2640 v3) processors and 128GB RAM, connected via a 10Gbps Ethernet (this configuration is described in [75] and by personal communication is that used in [110]), which is almost identical to our configuration described above. The number that we provide in Table 3.1 for this work is when utilizing 16 cores, and thus this is an almost identical configuration as Sharemind [110] (with 20 cores we achieve 1,324,117 AES operations per second). Observe that our latency (response time) is 70% of [118] and we achieve a throughput that is *14 times faster* than [110] (and so over an order of magnitude improvement). In fact, using a single core and a 1Gbps connection, we achieve approximately 100,000 AES operations per second (and latency of only 129ms); thus we can outperform the best Sharemind results on a very basic setup.

Table 3.1: Reported times for semi-honest 3-party computation & honest majority; the throughput is measured in **AES computations per second** (the last two rows with similar configurations).

Year	Ref.	Latency	Throughput
2010	[41]	2000s	-
2012	[80]	14.28ms	320
2013	[81]	323ms	3450
2016	[118, Table 5.3]	223ms	25,000
2016	[110]	-	90,000
2016	this work	166ms	1,242,310

We remark that other work on GCs (e.g., two-party Yao with semi-honest adversaries) achieves much lower latency (e.g., 16ms reported in [62]). However, each garbled AES circuit is of size at least 1.3Mb (using the latest half-gates optimization [127]), not taking into account additional messages that are sent. It is therefore physically impossible to go beyond 7500 AES computations per second on a 10Gbps network (where we achieve 1.4 million). In addition, the two-party GMW approach using efficient oblivious transfer (OT) extensions is blocked by the speed of the OTs (with two OTs required per gate). Considering the communication bottleneck, each OT requires transmitting a minimum of 128 bits. Thus, the communication is approximately the same as with a GC. (The fastest known implementation [69] can process 5 million OTs per second on a 1Gbps network giving under 500 AES computations per second. This is not far from optimal assuming linear scale-up on a 10Gbps network.) Of course, we require an additional server, in contrast to the Yao and GMW protocols.

3.3 The New Communication-Efficient Protocol for 3-Party Computation

In this section, we describe our new protocol for three parties. Our protocol works for arithmetic circuits over the ring modulo 2^n with Boolean circuits being a special case (with $n = 1$). The protocol uses only very simple ring addition and multiplication operations, which in the Boolean case reduces simply to bitwise AND and XOR. In

addition, the protocol has very low communication: a single ring element is sent per multiplication gate and there is no communication for addition gates. In the Boolean case, we therefore have that the only communication is a single bit per AND gate.

Correlated randomness. Our protocol assumes that for every multiplication gate the three parties P_0, P_1, P_2 are given *correlated randomness* in the form of random ring elements x_0, x_1, x_2 under the constraint that $x_0 + x_1 + x_2 = 0$. We show how this can be achieved in practice with great efficiency using AES. (Thus, our protocol is information-theoretically secure with perfect correlated randomness, but the actual implementation is computationally secure due to the use of AES to generate the correlated randomness.)

3.3.1 Securely Computing Boolean Circuits

In order to simplify the exposition, we begin by describing the protocol for the special case of Boolean circuits with AND and XOR gates.

Protocol 3.1 : Sharing input to the parties

- **Inputs:** A dealer (one of three parties or client) holds a bit v
 - **The protocol:**
 1. The dealer run the secret sharing protocol in Definition 2.3.2 and obtain $\mathbf{Share}(\mathbb{Z}_2, v) = ([x]_0^2, [x]_1^2, [x]_2^2)$.
 2. The dealer set the shares for each party as follows:
 - P_0 's share is $[x]_0^2 = ([x]_{0,0}^2, [x]_{0,1}^2) = (x_2 \oplus x_0, x_0)$.
 - P_1 's share is $[x]_1^2 = ([x]_{1,0}^2, [x]_{1,1}^2) = (x_0 \oplus x_1, x_1)$.
 - P_2 's share is $[x]_2^2 = ([x]_{2,0}^2, [x]_{2,1}^2) = (x_1 \oplus x_2, x_2)$.
 3. The dealer send $[x]_i^2$ to the party P_i where $i \in \{0, 1, 2\}$.
-

Recall that no single party's share reveals anything about v . In addition, any two shares suffice to obtain v ; e.g., given $[x]_0^2, [x]_1^2$, we can compute $v = [x]_{0,0}^2 \oplus [x]_{1,1}^2 = (x_2 \oplus x_0) \oplus x_1$.

Protocol 3.2 : Computing XOR gate

- **Inputs:** Each party $P_i (i \in \{0, 1, 2\})$ has the shares $[x]_i^2, [y]_i^2$ for secrets x and y , and the opcode **add** for addition.
- **The protocol:**
 1. For each $i \in \{0, 1, 2\}$, P_i generates

$$[z]_i^2 := ([x]_{i,0}^2 \oplus [y]_{i,0}^2, [x]_{i,1}^2 \oplus [y]_{i,1}^2)$$

and outputs **(add, $[z]_i^2$)**.

This operation as the whole is denoted by $[z]^2 = \mathbf{add}([x]^2, [y]^2)$.

XOR (addition) gates. Now we assume that each party shares $[x]^2$ and $[y]^2$ where $x, y \in \{0, 1\}$, and the parties want to obtain the shares of $[x + y]^2$. We describe the MPC protocol for XOR (addition) gate in Protocol 3.2.

In order to compute a secret sharing of $x + y$, each P_i locally computes the shares of $[x]^2$ and $[y]^2$ (no communication is needed).

We can easily check that $[z]^2$ is a valid share of $x + y$, since $[z]_{i,0}^2 + [z]_{i+1,0}^2 = ([x]_{i,0}^2 + [y]_{i,0}^2) + ([x]_{i+1,0}^2 + [y]_{i+1,0}^2) = ([x]_{i,0}^2 + [x]_{i+1,0}^2) + ([y]_{i,0}^2 + [y]_{i+1,0}^2) = x + y$ for all $i \in \{0, 1, 2\}$.

AND (multiplication) gates. We now show how the parties can compute AND (equivalently, multiplication) gates; this subprotocol requires each party to send a single bit only. The protocol works in two phases: in the first phase the parties compute a simple $(3, 3)$ XOR-sharing of the AND of the input bits, and in the second phase they convert the $(3, 3)$ -sharing into the above-defined $(2, 3)$ -sharing.

We describe the MPC protocol for computing $(2, 3)$ -shares of $x \cdot y = x \wedge y$ in Protocol 3 (from here on, we will denote multiplication of a and b by simply ab). Here we assume that the parties P_0, P_1, P_2 are able to obtain random $\alpha_0, \alpha_1, \alpha_2 \in \{0, 1\}$ such that $\alpha_0 \oplus \alpha_1 \oplus \alpha_2 = 0$. We will explain how to obtain such correlated randomness

in Section 3.3.2.

Protocol 3.3 : Computing AND gate

- **Inputs:** Each party $P_i (i \in \{0, 1, 2\})$ has the shares $[x]_i^2, [y]_i^2$ for secrets x and y , and the opcode **mult** for multiplication.
- **Auxiliary Input:** We assume that the parties P_i hold *correlated randomness* α_i , respectively, where $\alpha_0 \oplus \alpha_1 \oplus \alpha_2 = 0$.
- **The protocol:**

1. For each $i \in \{0, 1, 2\}$, P_i generates

$$w_i = [x]_{i,0} \cdot [y]_{i,0} \oplus [x]_{i,1} \cdot [y]_{i,1} \oplus \alpha_i,$$

where “ \cdot ” is the multiplication over \mathbb{Z}_2 , and sends $(\mathbf{mult_msg}, w_i)$ to P_{i+1} .

2. For each $i \in \{0, 1, 2\}$, P_i generates

$$[z]_i^2 := (w_{i-1} \oplus w_i, w_{i-1})$$

and outputs $(\mathbf{mult}, [z]_i^q)$.

This operation as the whole is denoted by $[z]^2 = \mathbf{mult}([x]^2, [y]^2)$.

For the correctness, we recall $x = x_0 + x_1 + x_2 \pmod q$, $y = y_0 + y_1 + y_2 \pmod q$ and $z = x \cdot y = (x_0 + x_1 + x_2)(y_0 + y_1 + y_2)$. w_i ($i \in \{0, 1, 2\}$) at the Step 1 can be represented as $w_0 = x_0y_0 + x_2y_0 + x_0y_2 + \alpha_0$, $w_1 = x_1y_1 + x_1y_0 + x_0y_1 + \alpha_1$, $w_2 = x_2y_2 + x_2y_1 + x_1y_2 + \alpha_2$, and we can see $z = w_0 + w_1 + w_2 \pmod q$. Therefore, the share of Step 2 satisfies the form of RSSS described in Sect. 2.3.

The above explanation shows that the gate computation “works” in the sense that the invariant of the format of the shares is preserved after every gate is computed. The fact that the protocol is secure is proved later in Section 3.4.

The protocol. The full 3-party protocol works in the natural way. The parties first share their inputs using the secret sharing. They then compute each XOR and AND gate in the circuit according to a predetermined topological ordering for the circuit. Finally, the parties reconstruct their output on the output wires. (In the client/server model, external clients send the three parties sharings of their input according, and the three parties then compute the circuit in the same way on the shares received.)

Observe that each party communicates with exactly one other party only. This property also holds for the protocol of Sharemind [17, 18]. However, our secret-sharing scheme and multiplication protocol are completely different.

3.3.2 Generating Correlated Randomness

Our protocol relies on the fact that the parties hold random bits $\alpha, \beta, \gamma \in \{0, 1\}$ such that $\alpha_0 \oplus \alpha_1 \oplus \alpha_2 = 0$ for every AND gate. In this section, we show how the parties can efficiently generate such $\alpha_0, \alpha_1, \alpha_2$.

Information-theoretic correlated randomness. It is possible to securely generate correlated randomness with perfect security by having each party P_i simply choose a random $\rho_i \in \{0, 1\}$ and send it to P_{i+1} (where P_2 sends to P_0). Then, each party takes its random bit to be the XOR of the bit it chose and the bit it received: P_0 computes $\alpha_0 = \rho_2 \oplus \rho_0$, P_1 computes $\alpha_1 = \rho_0 \oplus \rho_1$ and $\alpha_2 = \rho_1 \oplus \rho_2$. Observe that $\alpha_0 + \alpha_1 + \alpha_2 = 0$ as required. In addition, if P_0 is corrupted, then it knows nothing about α_1 and α_2 except that $\alpha_1 \oplus \alpha_2 = \alpha_0$. This is because α_1 and α_2 both include ρ_1 in their computation and this is unknown to P_0 . A similar argument holds for a corrupted P_1 or P_2 . Despite the elegance and simplicity of this solution, we use a different approach. This is due to the fact that this would *double* the communication per AND gate; it is true that this is still very little communication. However, given that communication is the bottleneck, it would halve the throughput.

Computational correlated randomness. We now show how it is possible to securely compute correlated randomness *computationally* without *any interaction* beyond a short initial setup. This enables us to maintain the current situation where parties need only transmit a single bit per AND gate. This method is similar to that of the PRSS subprotocol in [37], but simpler since Shamir sharing is not needed.

Protocol 3.4 : Generating computational correlated randomness

- **Inputs:** Let κ be the security parameter, and let $F : \{0, 1\}^\kappa \times \{0, 1\}^\kappa \rightarrow \{0, 1\}$ be a pseudorandom function outputting a single bit, and a unique identifier $id \in \{0, 1\}^\kappa$ which corresponds to the AND gate to be computed.
 - **The protocol:**
 - **Init:**
 1. Each P_i chooses a random $k_i \in \{0, 1\}^\kappa$.
 2. Party P_0 sends k_0 to P_2 , party P_1 sends k_1 to P_0 and party P_2 sends k_2 to P_1 .

P_0 holds k_0, k_1 , P_1 holds k_1, k_2 and P_2 holds k_2, k_0 .
 - **GetNextBit:** Given a unique identifier $id \in \{0, 1\}^\kappa$,
 1. P_0 computes $\alpha_0 = F_{k_0}(id) \oplus F_{k_1}(id)$.
 2. P_1 computes $\alpha_1 = F_{k_1}(id) \oplus F_{k_2}(id)$.
 3. P_2 computes $\alpha_2 = F_{k_2}(id) \oplus F_{k_0}(id)$.
-

Observe that $\alpha_0 + \alpha_1 + \alpha_2 = 0$. Furthermore, P_0 does not know k_2 which is used to generate α_1 and α_2 . Thus, α_1 and α_2 are pseudorandom to P_0 , under the constraint that $\alpha_1 \oplus \alpha_2 = \alpha_0$. In practice, the id can be a counter that all parties locally increment at every call to **GetNextBit**.

3.3.3 The Ring with General Modulus: 2^n and Fields

Our protocol above works for Boolean circuits. However, in some cases arithmetic circuits are far more efficient, for example over 2^n and arbitrary fields of size greater than 2. Fortunately, our protocol can be easily extended to the cases of arithmetic operations. In this section, we show how to generalize the protocol above. In the following, we describe MPC protocols for the ring modulo q where q is an positive integer.

We remark that when taking $q = 2$ we have that addition (and subtraction) is the same as XOR, and multiplication is the same as AND. In this case, the protocol here is exactly that described in Section 3.3.1.

Addition gates. As in the Boolean case, addition gates are computed by locally adding the shares modulo 2^n . To confirm this, we describe the general case of the addition in Protocol 5.

Protocol 3.5 : Computing arithmetic addition gate over \mathbb{Z}_q

- **Inputs:** Each party $P_i (i \in \{0, 1, 2\})$ has the shares $[x]_i^q, [y]_i^q$ for secrets x and y over \mathbb{Z}_q , and the opcode **add** for addition.

- **The protocol:**

1. For each $i \in \{0, 1, 2\}$, P_i generates

$$[z]_i^q := ([x]_{i,0}^q \oplus [y]_{i,0}^q, [x]_{i,1}^q \oplus [y]_{i,1}^q)$$

and outputs $(\mathbf{add}, [z]_i^q)$.

This operation as the whole is denoted by $[z]^q = \mathbf{add}([x]^q, [y]^q)$.

Multiplication gates: The MPC for multiplication gate also can be constructed in similar way to AND gate. It is necessary to pay attention only to the sign during the operation. We explicitly describe the MPC protocol for multiplication over 2^n in Protocol 6.

Protocol 3.6 : Computing arithmetic multiplication gate

- **Inputs:** Each party $P_i (i \in \{0, 1, 2\})$ has the shares $[x]_i^q, [y]_i^q$ for secrets x and y , and the opcode **mult** for multiplication.
- **Auxiliary Input:** We assume that the parties P_i hold *correlated randomness* α_i , respectively, where $\alpha_0 \oplus \alpha_1 \oplus \alpha_2 = 0$.
- **The protocol:**

1. For each $i \in \{0, 1, 2\}$, P_i generates

$$w_i = [x]_{i,0}^q \cdot [y]_{i,0}^q - [x]_{i,1}^q \cdot [y]_{i,1}^q + \alpha_i,$$

where “ \cdot ” is the multiplication over \mathbb{Z}_2 , and sends $(\mathbf{mult_msg}, w_i)$ to P_{i+1} .

2. For each $i \in \{0, 1, 2\}$, P_i generates

$$[z]_i^q := (w_{i-1} + w_i, w_{i-1})$$

and outputs $(\mathbf{mult}, [z]_i^q)$.

This operation as the whole is denoted by $[z]^q = \mathbf{mult}([x]^q, [y]^q)$.

Generating correlated randomness. The parties use the same (computational) method as described in Section 3.3.2, with the following differences. First, we assume that F_k is a pseudorandom function mapping strings into \mathbb{Z}_q . Second, party P_0 computes $\alpha = F_{k_0}(id) - F_{k_1}(id)$, party P_1 computes $\beta = F_{k_1}(id) - F_{k_2}(id)$, and party P_2 computes $\gamma = F_{k_2}(id) - F_{k_0}(id)$.

3.3.4 Protocol Efficiency and Comparison

In the case of arbitrary finite fields, Shamir’s secret-sharing [112] is “ideal”, meaning that the size of the share equals the size of the secret (which is minimum size), as long

as the number of parties is less than the size of the field. In our protocol, the secret sharing is not ideal since it consists of two ring or field elements instead of a single field element. However, this is of little consequence when considering the efficiency of the protocol since our protocol requires only sending a *single* element per multiplication gate. In addition, the computation consists merely of two multiplications and two additions.

In comparison, the BGW protocol [14, 8] requires transmitting two field elements per multiplication gate by each party when using [109] method (with a single round of communication). In addition, when considering Boolean circuits, at least two bits are needed per field element, since there are 3 parties. Furthermore, the computation requires polynomial evaluations which are far more expensive.

In the Sharemind protocol [17, 18], the parties transmit five ring elements per AND gate over two communication rounds, and compute 3 multiplications and 8 additions. We remark that our method for generating correlated randomness can be used to reduce the number of elements sent in the Sharemind protocol from 5 to 2 and to reduce the number of communication rounds to 1.

3.4 Security against Semi-Honest Adversaries

In this section, we prove that our protocol is secure in the presence of one semi-honest adversarial party (in Section 3.5 we prove that the protocol is private in the presence of one malicious adversary). Semi-honest security is sufficient when parties somewhat trust each other, but are concerned with inadvertent leakage or cannot share their raw information due to privacy regulations. It is also sufficient in cases where it is reasonable to assume that the parties running the protocol are unable to replace the installed code. Nevertheless, security against covert or malicious adversaries is preferable, providing far higher guarantees; we leave extensions of our protocol to these settings for future work.

Since the protocol for Boolean circuits is a *special case* of the protocol for the ring modulo q , we prove the security for the case of the ring modulo q . The proof is identical in the case of fields with more than 3 elements.

Proof outline. We denote a protocol π in the g -hybrid model by π^g , and the real protocol obtained by replacing calls to g by invocations of subprotocol ρ by π^ρ . We abuse notation and write $\pi^g \equiv f$ to say that π securely computes f in the g -hybrid

model, and write $\pi^\rho \equiv f$ to say that the real protocol π^ρ securely computes f . Denote by σ the protocol that computes the correlated randomness functionality \mathcal{F}_{CR} , by ρ the protocol that computes the multiplication functionality $\mathcal{F}_{\text{MULT}}$ in the \mathcal{F}_{CR} -hybrid model, and by π the protocol that computes the functionality f in the $\mathcal{F}_{\text{MULT}}$ -hybrid model. Our goal is to prove that π^{ρ^σ} securely computes f in the presence of one static semi-honest corrupted party.

Let f be a 3-ary functionality. We begin by proving that $\pi^{\mathcal{F}_{\text{MULT}}}$ computes f with *perfect* security in the presence of one static semi-honest party. Next, we prove that $\rho^{\mathcal{F}_{\text{CR}}}$ computes $\mathcal{F}_{\text{MULT}}$ with *perfect* security in the presence of one static semi-honest party in the \mathcal{F}_{CR} -hybrid model. Finally, we prove that σ computes \mathcal{F}_{CR} with *computational* security in the presence of one static semi-honest party. The reason for achieving only computational security for the correlated randomness protocol is that we use a pseudorandom function to compute the random values. The proof in this case, thereby, works by making a reduction to a distinguisher between a pseudorandom function and a random function.

Once we have proved that $f \equiv \pi^{\mathcal{F}_{\text{MULT}}}$, that $\mathcal{F}_{\text{MULT}} \equiv \rho^{\mathcal{F}_{\text{CR}}}$ and that $\mathcal{F}_{\text{CR}} \stackrel{c}{\equiv} \sigma$, we can apply the composition theorem of [29] (using the fact that universal composability is implied via [78]) to conclude that $\pi^{\rho^{\mathcal{F}_{\text{CR}}}} \equiv f$; that is, π^{ρ^σ} computes f with computational security in the presence of one static semi-honest adversary.

3.4.1 Computing f in the $\mathcal{F}_{\text{mult}}$ -Hybrid Model

We define the multiplication functionality $\mathcal{F}_{\text{MULT}}$ that receives input shares of two values v_a, v_b as input and outputs shares of the product $v_a v_b$, according to the secret-sharing scheme described in Section 3.3.3. Intuitively, $\mathcal{F}_{\text{MULT}}$ should be defined by receiving the shares of all parties, reconstructing the values v_0, v_1 from the shares, and then generating a random resharing of the $v_0 v_1$. Indeed, if secure coin tossing were used instead of the method that we use for correlated randomness, then $\mathcal{F}_{\text{MULT}}$ would be defined in this natural way. However, this would require additional communication and would affect performance. We therefore need to define a more complex multiplication functionality. In order to understand why this is needed, recall the real protocol and consider the specific case that P_1 is corrupted. In order to simplify this explanation, consider the Boolean case.

Functionality 3.1 : $\mathcal{F}_{\text{MULT}}$ – multiplication

Let $F : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \mathbb{Z}_{2^n}$ be a keyed function. Upon invocation, \mathcal{F}_{CR} chooses a pair of keys $k, k' \in \{0, 1\}^\kappa$ and sends them to the adversary controlling party P_i . Then:

1. $\mathcal{F}_{\text{MULT}}$ receives $(([v_a]_{j,0}, [v_a]_{j,1}), ([v_b]_{j,0}, [v_b]_{j,1}))$ from each P_j and receives a pair $([z]_{i,0}, [z]_{i,1}) \in \mathbb{Z}_{2^n} \times \mathbb{Z}_{2^n}$ from the adversary controlling P_i .
 2. $\mathcal{F}_{\text{MULT}}$ computes $v_a = [v_a]_{0,0} + [v_a]_{1,1}$ and $v_b = [v_b]_{1,0} + [v_b]_{2,1}$ and $v_c = v_a v_b$.
 3. $\mathcal{F}_{\text{MULT}}$ sets $[z]_{i-1,0} = v_c - [z]_{i,1}$ and $[z]_{i+1,0} = -[z]_{i,0} - [z]_{i-1,0}$, and sets $[z]_{i-1,1} = v_c - [z]_{i+1,0}$ and $[z]_{i+1,1} = v_c - [z]_{i,0}$.
 4. $\mathcal{F}_{\text{MULT}}$ sends each P_j the pair $([z]_{j,0}, [z]_{j,1})$ (for $j \in \{0, 1, 2\}$).
-

Party P_0 computes $w_i = [v_a]_{0,0} \cdot [v_b]_{0,0} \oplus [v_a]_{0,1} \cdot [v_b]_{0,1} \oplus \alpha_0$ and receives w_2 from P_2 . Observe that α_0 is not random to the corrupted P_0 and is fixed by a very specific computation (specifically, $F_{k_0}(\text{id}) \oplus F_{k_1}(\text{id})$; see Section 3.3.2). Thus, P_0 's computation of w_0 is *deterministic*. Now, P_0 's output from the multiplication protocol is the pair $([z]_{0,0}, [z]_{0,1})$ where $[z]_{0,0} = w_0 \oplus w_2$ and $[v_a v_b]_{0,1} = w_0$. Since w_2 is received from P_2 and is masked with the correlated randomness that P_2 receives (which is generated using a pseudorandom function with a key not known to P_0) this value is random. However, $[z]_{0,1}$ is *fixed* (since it equals w_0). Stated differently, given that w_0 is fixed, there are exactly two possible values for $([z]_{0,0}, [z]_{0,1})$ based on $[z]_{0,0} = 0$ or $[z]_{0,0} = 1$. In contrast, a random secret sharing has four possible values for $([z]_{0,0}, [z]_{0,1})$, with all four combinations of $[z]_{0,0}, [z]_{0,1} \in \{0, 1\}$. Thus, it is *not* true that the multiplication protocol generates a new random sharing of the product.

In order to solve this problem, we take a different approach. We allow the corrupted party to completely determine its share $([z]_{0,0}, [z]_{0,1})$. The functionality $\mathcal{F}_{\text{MULT}}$ then determines the other parties' shares based on $([z]_{0,0}, [z]_{0,1})$ and the product $v_a v_b$. Interestingly, in this secret sharing, a single share together with the secret

fully determines all other shares. This is because each $[z]_{i,1} = v_a v_b - [z]_{i-1,0}$. Thus, $([z]_{0,0}, [z]_{0,1})$ and $v_a v_b$ determines $[z]_{i-1,0} = v_a v_b - [z]_{i,1}$, which in turn determines z_{i+1} since $[z]_{0,0} + [z]_{1,0} + [z]_{2,0} = 0$. Finally, all z values together with $v_a v_b$ determine all c values.

We denote the protocol for securely computing f that is defined in Section 3.3.3 by Protocol 3.3.3. We now prove the security of Protocol 3.3.3 according to Definition 2.4.2.

Theorem 3.4.1. *Let $f : ((\mathbb{Z}_{2^n})^*)^3 \rightarrow ((\mathbb{Z}_{2^n})^*)^3$ be a 3-ary functionality. Then, Protocol 3.3.3 computes f with perfect security in the $\mathcal{F}_{\text{MULT}}$ -hybrid model, in the presence of one semi-honest corrupted party.*

Proof. Since the circuit C computes functionality f the first requirement of definition 2.4.2 is immediately fulfilled. We now proceed to the second requirement of the definition.

Let P_i be the corrupted party. Our Simulator \mathcal{S} is invoked upon P_i 's input, \vec{v}^i and P_i 's output, $f_i(\vec{v})$. The simulator \mathcal{S} needs to output a transcript that is identically distributed to the view of P_i and therefore consists of $(\vec{v}^i, r^i, m_{\text{input}}^i, m_1^i, \dots, m_\ell^i, m_{\text{output}}^i)$ where r^i is P_i 's random tape, m_{input}^i is the vector of shares that are sent to P_i at the input sharing stage, ℓ is the number of multiplication gates in the circuit (since we have interaction only in these gates) and m_k^i is the message P_i received when computing multiplication gate G_k (recall that in our protocol, each party receives only one message per each multiplication gate from the F_{mult} ideal functionality), and m_{output}^i is the vector of shares that are sent to P_i at the output reconstruction stage. Thus, we denote

$$\text{VIEW}_i^\pi(\vec{v}) = (\vec{v}^i, r^i, m_{\text{input}}^i, m_1^i, \dots, m_\ell^i, m_{\text{output}}^i) \quad (2)$$

Next, we describe our simulator \mathcal{S} .

$\mathcal{S}(\vec{v}^i, f_i(\vec{v}))$:

1. *Simulating the input sharing stage:*

- (a) For party P_i , the simulator \mathcal{S} chooses a uniformly distributed random tape r^i . The random tape fully determines, for each input value $v_k^i \in \mathbb{Z}_{2^n}$, the random values of $x_{k,1}^i, x_{k,2}^i, x_{k,3}^i \in \mathbb{Z}_{2^n}$ such that $x_{k,1}^i + x_{k,2}^i + x_{k,3}^i =$

- 0 mod 2^n . Then, for each $v_k^i \in \vec{v}^i$, \mathcal{S} defines P_i 's share of v_k^i to be the pair $(x_{k,i}^i, x_{k,i-1}^i - v_k^i)$, where $x_{k,i-1}^i = x_{k,3}^i$ when $i = 1$.
- (b) Let U be the set of input wires. For every input wire $k \in U$ associated with party P_j where $j \neq i$, \mathcal{S} chooses uniformly $y_{k,1}^j, y_{k,2}^j$ from \mathbb{Z}_{2^n} .
- (c) The simulator \mathcal{S} sets P_i 's view at this stage to be $\{\{(y_{k,1}^j, y_{k,2}^j)\}_{k \in U, j \in \{0,1,2\} \setminus \{i\}}\}$.
2. *Simulating the circuit emulation stage:* For every gate G_k in the circuit in topological order:
- (a) *If G_k is an addition gate:* Let $(y_{k,1}^i, y_{k,2}^i)$ and $(z_{k,1}^i, z_{k,2}^i)$ be the two pair of shares of the gate's input wires held by party P_i . Then, \mathcal{S} defines P_i 's shares of the output wire to be $(y_{k,1}^i + z_{k,1}^i, y_{k,2}^i + z_{k,2}^i)$.
- (b) *If G_k is a multiplication-by-a-constant gate with a constant c :* Let $(y_{k,1}^i, y_{k,2}^i)$ be the pair of shares of the gate's input wires held by party P_i . Then, \mathcal{S} defines P_i 's shares of the output wire to be $(cy_{k,1}^i, cy_{k,2}^i)$.
- (c) *If G_k is a multiplication gate:* The simulator \mathcal{S} chooses $z_{k,1}^i, z_{k,2}^i$ uniformly at random from \mathbb{Z}_{2^n} , and defines the output wire shares of P_i to be $(z_{k,1}^i, z_{k,2}^i)$. Then, \mathcal{S} adds the shares to the view of the corrupted party P_i .
3. *Simulating the output reconstruction stage:* Let \vec{o}^i be the circuit's output wires that are associated with P_i . For each output wire $o_k^i \in \vec{o}^i$, let $(y_{k,1}^i, y_{k,2}^i)$ be the share of P_i on this wire. Since the simulator \mathcal{S} holds P_i 's output $f_i(\vec{v})$, it knows the actual value v_k^i that is on the output wire o_k^i . Thus, \mathcal{S} sets $x_{k,i}^i = y_{k,1}^i$ and $x_{k,i-1}^i = y_{k,2}^i + v_k^i$. (Thus, the share of P_i on o_k^i is the pair $(x_{k,i}^i, x_{k,i-1}^i - v_k^i)$). Then, \mathcal{S} computes $x_{k,i+1}^i = 0 - x_{k,i}^i - x_{k,i-1}^i \bmod \mathbb{Z}_{2^n}$ and sets the share of P_{i-1} to be $(x_{k,i-1}^i, x_{k,i-2}^i - v_k^i)$, and the share of P_{i+1} to be $(x_{k,i+1}^i, x_{k,i}^i - v_k^i)$. Thus, for each output wire $o_k^i \in \vec{o}^i$, \mathcal{S} adds the shares of the other parties (i.e, P_{i-1} and P_{i+1}) that it computed to P_i 's view and halts.

Now, we show that the view of the corrupted party generated by the simulator is identical to the view of the corrupted party when using the real execution of the protocol. We prove this in two steps. First, denote by $\widetilde{\text{VIEW}}_i^\pi(\vec{v})$ the partial view of the corrupted party up to the output reconstruction stage (and not including that

stage). Likewise, denote by $\tilde{\mathcal{S}}_i(\vec{v}^i, f_i(\vec{v}^i))$ the partial view generated by the simulator up to but not including the output reconstruction stage.

Claim 3.4.2. *For every $\vec{v} \in ((\mathbb{Z}_{2^n})^*)^3$ and every $i \in \{0, 1, 2\}$,*

$$\left\{ \widetilde{\text{VIEW}}_i^\pi(\vec{v}) \right\} \equiv \left\{ \tilde{\mathcal{S}}(\vec{v}^i, f_i(\vec{v})) \right\} \quad (3)$$

Proof. Note that the view of P_i is actually a set of shares (where each share is a pair of values) and that the only difference between the two partial views is that the view generated in a real execution consists of *random* shares of the correct values that are on the circuit wires, whereas the view generated by the simulator consists of *random* shares of the value '0' (Observe that in both cases the share of P_i at the end of each multiplication gate is truly random, meaning that its values are *independent* of the shares generated so far. Specifically, in the real execution the parties receive a random independent share from the ideal functionality, while in the simulation, \mathcal{S} chooses two random independent values and set them to be the output wire's share of the corrupted party). Hence we have that the two set of shares are identically distributed and therefore the two partial views are also identically distributed as required. ■

It remains to show that the view generated by the simulator after the output reconstruction stage is identical to the view of the corrupted party in a real execution. For simplicity, we assume that the output wires appear immediately after multiplication gates (otherwise, they are fixed function of these values).

In order to prove the above, we prove that the *process* carried out by the simulator in the output reconstruction stage yields the same distribution as in the real protocol execution. We start by describing two processes, prove that they yield the same distribution and later show that these two processes are exactly the processes carried out by the simulator and in the real execution.

Random variable $X(s)$

- (1) Choose $x_0, x_1, x_2 \in_R \mathbb{Z}_{2^n}$ s.t. $\sum_{i=1}^3 x_i = 0 \bmod 2^n$
- (2) -
- (3) -
- (4) Output $\{(x_0, s - x_2), (x_1, s - x_0), (x_2, s - x_1)\}$

Random Variable $Y(s)$

- (1) Choose $x_i, y_i \in_R \mathbb{Z}_{2^n}$ for some $i \in \{0, 1, 2\}$
- (2) Set $x_{i-1} = s + y_i \bmod 2^n$ *
- (3) Set $x_{i+1} = 0 - x_i - x_{i-1} \bmod 2^n$ *
- (4) Output $\{(x_0, s - x_2), (x_1, s - x_0), (x_2, s - x_1)\}$

Claim 3.4.3. *For every $s \in \mathbb{Z}_{2^n}$, it holds that $\{X(s)\} \equiv \{Y(s)\}$*

Proof. In order to show that the distributions are identical we need to show that for all $j \in \{0, 1, 2\}$, x_j generated in $X(S)$'s process and x_j generated in $Y(s)$'s process are identically distributed. To see this, we first look at a slightly modified process which generates a random variable $X'(s)$.

Random variable $X'(s)$

- (1) Choose $x_i \in_R \mathbb{Z}_{2^n}$ for some $i \in \{0, 1, 2\}$
- (2) Choose $x_{i-1} \in_R \mathbb{Z}_{2^n}$ *
- (3) Set $x_{i+1} = 0 - x_i - x_{i-1} \bmod 2^n$ *
- (4) Output $\{(x_0, s - x_2), (x_1, s - x_0), (x_2, s - x_1)\}$

*where $x_{i-1} = x_3$ when $i = 1$, and $x_{i+1} = x_1$ when $i = 3$

The only difference between the process of generating $X'(s)$ and the process of generating $Y(s)$ is in step (2) where x_{i-1} 's value is determined. In the generating process $X'(s)$, x_{i-1} is chosen uniformly from \mathbb{Z}_{2^n} at random, whereas in the generating process $Y(s)$, x_{i-1} is set to be the sum of the secret value s and a uniform random variable y_i over \mathbb{Z}_{2^n} that appears nowhere else in the process. Therefore, x_{i-1} is also a uniform random variable over \mathbb{Z}_{2^n} and thus we conclude that $\{X'(s)\} \equiv \{Y(s)\}$.

Next, observe that in the process of generating $X'(s)$, x_0, x_1, x_2 are chosen uniformly from \mathbb{Z}_{2^n} at random such that $x_0 + x_1 + x_2 = 0 \bmod 2^n$, exactly as in the process of generating $X(s)$. Therefore, we can conclude that $\{X(s)\} \equiv \{X'(s)\}$.

Combining the fact that $\{X(s)\} \equiv \{X'(s)\}$ with the fact that $\{X'(s)\} \equiv \{Y(s)\}$, we obtain that $\{X(s)\} \equiv \{Y(s)\}$ as required. ■

We now use Claim 3.4.3 to prove:

Claim 3.4.4. *If $\{\widetilde{\text{VIEW}}_i^\pi(\vec{v})\} \equiv \{\tilde{\mathcal{S}}(\vec{v}^i, f_i(\vec{v}))\}$, then $\{\text{VIEW}_i^\pi(\vec{v})\} \equiv \{\mathcal{S}(\vec{v}^i, f_i(\vec{v}))\}$*

Proof. At the beginning of the output reconstruction stage, the corrupted party P_i holds a pair of values on each of its output wires. In the simulator procedure, these values are just pairs of random values from \mathbb{Z}_{2^n} , while in the execution of the protocol these values are correct shares of the actual value that is on the output wire.

Since in the real execution all the parties hold a correct share of the value that is on an output wire, the view of the corrupted party in the output reconstruction stage is exactly the output of the process $X(s)$ where s is the value on the output wire.

In contrast, in the simulation, the corrupted party holds a pair of random values and then the simulator computes the share of the other parties based on these values. This is done exactly as in the process of $Y(s)$.

Thus, if $\{\widetilde{\text{VIEW}}_i^\pi(\vec{v})\} \equiv \{\tilde{\mathcal{S}}(\vec{v}^i, f_i(\vec{v}))\}$, from Claim 3.4.3 we obtain that $\{\text{VIEW}_i^\pi(\vec{v})\} \equiv \{\mathcal{S}(\vec{v}^i, f_i(\vec{v}))\}$ as required. ■

Combining claims 3.4.2 and claim 3.4.4 we have that $\{\text{VIEW}_i^\pi(\vec{v})\} \equiv \{\mathcal{S}(\vec{v}^i, f_i(\vec{v}))\}$ as required. ■

3.4.2 Computing $\mathcal{F}_{\text{mult}}$ in the \mathcal{F}_{cr} -Hybrid Model

In this section, we prove that the multiplication protocol described in Section 3.3.3 computes the $\mathcal{F}_{\text{MULT}}$ functionality with perfect security in the presence of one semi-honest corrupted party. Recall that our protocol utilizes correlated randomness in the form of random $\alpha_0, \alpha_1, \alpha_2$ such that $\alpha_0 + \alpha_1 + \alpha_2 = 0$.

Functionality 3.2 : \mathcal{F}_{CR} – corr. randomness

Let $F : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \mathbb{Z}_{2^n}$ be a keyed function. Upon invocation, \mathcal{F}_{CR} chooses a pair of keys $k, k' \in \{0, 1\}^\kappa$ and sends them to the adversary controlling party P_i . Then:

- Upon receiving input id from all parties, functionality \mathcal{F}_{CR} computes $\alpha_i = F_k(id) - F_{k'}(id)$ and chooses random values $\alpha_{i-1}, \alpha_{i+1} \in \mathbb{Z}_{2^n}$ under the constraint that $\alpha_0 + \alpha_1 + \alpha_2 = 0 \pmod{2^n}$. \mathcal{F}_{CR} sends α_j to P_j for every j .
-

Background – correlated randomness. First, we formally define the ideal functionality \mathcal{F}_{CR} . A naive definition would be to have the ideal functionality choose $\alpha_0, \alpha_1, \alpha_2$ and send α_i to P_i for $i \in \{0, 1, 2\}$. However, securely realizing such a

functionality would require interaction (as in the information-theoretic method first described in Section 3.3.2). In order to model our computational method described in Section 3.3.2 (which is the same as used for the ring case) we need to take into account that the corrupted party's value is generated in a very specific way using a pseudorandom function. In order for the $\mathcal{F}_{\text{MULT}}$ protocol to be secure, all that is needed is that the corrupted party knows *nothing* about the honest party's values (beyond the given constraint that all values sum to zero). In particular, there is no requirement regarding how the corrupted party's value is generated.

Protocol 3.7 : Computing $\mathcal{F}_{\text{MULT}}$

- **Inputs:** Each party P_j (with $j \in \{0, 1, 2\}$) holds two pairs of values $(x_j, a_j), (y_j, b_j)$ which are valid $(2, 3)$ -sharings of the values that are on the input wires.
 - **Auxiliary input:** The parties hold the same unique identifier id (in the protocol using $\mathcal{F}_{\text{MULT}}$ this identifier can be the index of the multiplication gate being computed).
 - **The protocol:**
 1. **Correlated randomness:** Each party P_j (with $j \in \{0, 1, 2\}$) sends id to \mathcal{F}_{CR} and receives back α_j from \mathcal{F}_{CR} .
 2. **Local computation:** Each party P_j locally computes: $r_j = x_j y_j - a_j b_j + \alpha_j$.
 3. **Communication:** Party P_j sends r_j to party P_{j+1} (recall that $P_{j+1} = P_0$ when $j = 2$).
 - **Output:** Each P_j outputs (z_j, c_j) where $z_j = r_{j-1} + r_j$ and $c_j = r_j$; recall $r_{j-1} = r_2$ when $j = 0$.
-

Recall that in our protocol each party holds two keys which are used to locally

compute the correlated randomness. In order for the view of the corrupted party to be like in the real protocol, we define the functionality \mathcal{F}_{CR} so that it generates the corrupted party's value in this exact same way (i.e., $F_k(id) - F_{k'}(id)$ for keys k, k' ; see Section 3.3.3). As we have mentioned, the honest parties' values *are* chosen randomly, under the constraint that all values sum to zero.

The functionality is described formally in Functionality 3.2. The functionality chooses two keys k, k' for a pseudorandom function F and sends them to the corrupted party. We denote by κ the computational security parameter, and thus the length of the keys k, k' .

The multiplication protocol. A formal description of the protocol that securely computes the multiplication functionality $\mathcal{F}_{\text{MULT}}$ in the \mathcal{F}_{CR} -hybrid model appears in Protocol 3.7.

We now prove that the protocol is secure in the presence of one static semi-honest corrupted party.

Theorem 3.4.5. *Protocol 3.7 computes $\mathcal{F}_{\text{MULT}}$ with perfect security in the \mathcal{F}_{CR} -hybrid model in the presence of one semi-honest corrupted party.*

Proof. In the protocol, the corrupted party receives a single message. This message is an element from \mathbb{Z}_{2^n} which is uniformly distributed over \mathbb{Z}_{2^n} , due to the fact that each party masks its message using a random value received from the \mathcal{F}_{CR} functionality. Intuitively, the protocol is secure because all the corrupted party sees is a random element. (Note that the corrupted party also receives output from \mathcal{F}_{CR} but this is fully determined to be $\alpha_i = F_k(id) - F_{k'}(id)$.) We now prove this claim formally.

The $\mathcal{F}_{\text{MULT}}$ functionality as we have defined it is deterministic, and we therefore prove security via the simpler Definition 2.4.2. In order to show correctness, we need to show that the actual values $(z_0, c_0), (z_1, c_1), (z_2, c_2)$ output by all three parties from Protocol 3.7 are exactly the same values as those computed by $\mathcal{F}_{\text{MULT}}$. In order to see that this holds, recall that in Section 3.3.3 we showed that

$$z_0 + z_1 + z_2 = 0 \quad \text{and} \quad \forall j \in \{0, 1, 2\} \quad c_j = v_a v_b - z_{j-1}. \quad (4)$$

We claim that given a fixed (z_i, c_i) and $v_a v_b$, Eq. (4) implies that all values $z_{i-1}, c_{i-1}, z_{i+1}, c_{i+1}$ are *fully determined*. Specifically, let (z_i, c_i) be fixed and let $v_a v_b$ be the output value. Since for all $j \in \{0, 1, 2\}$ we have $c_j = v_a v_b - z_{j-1}$, this implies that

$z_{i-1} = v_a v_b - c_i$ is determined, which in turn determines $z_{i+1} = -z_i - z_{i-1}$. Finally, this determines $c_{i+1} = v_a v_b - z_i$ and $c_{i-1} = v_a v_b - z_{i+1}$. This is exactly the way that $\mathcal{F}_{\text{MULT}}$ computes the output values, and thus these are identical in the protocol and in the functionality output.

We now prove privacy by defining the simulator. The simulator \mathcal{S} receives the input and output of the corrupted party P_i from $\mathcal{F}_{\text{MULT}}$ as well as the auxiliary input id and (k, k') , and needs to compute the messages P_i sees during the execution. The input of the corrupted party P_i consists of two pair of shares $(x_i, a_i), (y_i, b_i)$ and it has no output. Intuitively, \mathcal{S} chooses a random element $r_{i-1} \in \mathbb{Z}_{2^n}$ and uses it to define the pair (z_i, c_i) that it sends to the trusted party computing $\mathcal{F}_{\text{MULT}}$. Formally, the simulator receives $((x_i, a_i), (y_i, b_i))$ and works as follows:

1. \mathcal{S} chooses a random $r_{i-1} \in \mathbb{Z}_{2^n}$.
2. \mathcal{S} sets $r_i = x_i y_i - a_i b_i + \alpha_i$ where $\alpha_i = F_k(id) - F_{k'}(id)$ as would be computed by \mathcal{F}_{CR} in the protocol.
3. \mathcal{S} sets $z_i = r_{i-1} + r_i$ and $c_i = r_i$.
4. \mathcal{S} sends (z_i, c_i) to $\mathcal{F}_{\text{MULT}}$.
5. \mathcal{S} adds α_i and r_{i-1} to the view of the corrupted party.

The values α_i and r_i are computed by \mathcal{S} exactly as by P_i in a real execution. The only difference is how r_{i-1} is computed; P_i receives $r_{i-1} = x_{i-1} y_{i-1} - a_{i-1} b_{i-1} + \alpha_{i-1}$ from P_{i-1} in a real execution, whereas \mathcal{S} chooses $r_{i-1} \in \mathbb{Z}_{2^n}$ uniformly at random in the simulation. The distribution over these two values is *identical* by the fact that \mathcal{F}_{CR} chooses $\alpha_{i-1}, \alpha_{i+1}$. Specifically, \mathcal{F}_{CR} chooses these at random under the constraint that $\alpha_0 + \alpha_1 + \alpha_2 = 0$. However, this is equivalent to choosing $\alpha_{i-1} \in \mathbb{Z}_{2^n}$ uniformly at random and then setting $\alpha_{i+1} = -\alpha_i - \alpha_{i-1}$. Now, since α_{i-1} is uniformly random, this implies that r_{i-1} is uniformly random (since it is independent of all other values used in the generation of r_{i-1}). Thus, the distribution over the real r_{i-1} received by P_i in the protocol execution and over the simulated r_{i-1} generated by \mathcal{S} is identical. This completes the proof. ■

3.4.3 Computing \mathcal{F}_{cr} in the Plain Model

In this section, we prove that our protocol securely computes the \mathcal{F}_{CR} functionality in the presence of one semi-honest corrupted party. We have already presented the

\mathcal{F}_{CR} functionality in Functionality 3.2. The protocol for computing it appears in Protocol 3.8.

Protocol 3.8 : Computing \mathcal{F}_{CR}

- **Auxiliary input:** Each party holds a security parameter κ , a description of a pseudorandom function $F : \{0, 1\}^\kappa \times \{0, 1\}^\kappa \rightarrow \mathbb{Z}_{2^n}$.
 - **Setup (executed once):**
 1. Each party P_j chooses randomly $k_j \in \{0, 1\}^\kappa$.
 2. Each party P_j sends k_j to party P_{j+1} .
 - **Generating randomness:** Upon input id , each party P_j computes $\alpha_j = F_{k_j}(id) - F_{k_{j-1}}(id)$ and **outputs** it.
-

Theorem 3.4.6. *If $F_k(\cdot)$ is a pseudorandom function, then Protocol 3.8 computes \mathcal{F}_{CR} with computational security in the plain model, in the presence of 1 semi-honest corrupted party.*

PROOF SKETCH: Since the functionality is probabilistic, we need to use Definition 2.4.1. Unlike the previous security proofs we have seen, the security of this protocol is computational and it relies on the assumption that F_k is a pseudorandom function. Thus, we will show that the ability to distinguish between the outputs in the real and ideal executions can be used to distinguish between the pseudorandom function and a truly random function, in contradiction to the assumption.

Let P_i be the corrupted party. We define the simulator \mathcal{S} who simulates P_i 's view. \mathcal{S} is invoked on the security parameter 1^κ and works as follows:

1. \mathcal{S} receives k, k' from \mathcal{F}_{CR} when it is first invoked (see Functionality 3.2).
2. \mathcal{S} sets the random tape of P_i (used by P_i to sample k_i) to be the key k received from \mathcal{F}_{CR} .

3. \mathcal{S} simulates the setup phase by writing the key k' as the key k_{i-1} received by P_i from P_{i-1} .
4. From this point on, every time that P_i receives id for input, \mathcal{S} sends it to the trusted party computing \mathcal{F}_{CR} . (P_i receives back α_i but this equals $F_k(id) - F_{k'}(id) = F_{k_i}(id) - F_{k_{i-1}}(id)$ and is known to P_i . Also, this value is computed locally by P_i in the protocol and not received. Thus, \mathcal{S} does not include it in P_i 's view.)

It is easy to see that the view generated by the simulator which consists of the P_i 's random tape and the incoming message k_{i-1} is distributed identically to its view in a real execution. However, this is not sufficient, as we need to prove indistinguishability of the *joint distribution* of both the corrupted party's view and the honest parties' outputs. Observe that in the real protocol execution, the honest parties' outputs are generated using the pseudorandom function, whereas in the ideal world they are chosen randomly by \mathcal{F}_{CR} .

Intuitively, the proof follows from the fact that both P_{i-1} and P_{i+1} generate their values using the pseudorandom function F with key k_{i+1} that is independent of k_i and k_{i-1} . Thus, replacing $F_{k_{i+1}}$ with a truly random function f results in P_{i-1} and P_{i+1} generating values α_{i-1} and α_{i+1} that are random under the constraint that $\alpha_0 + \alpha_1 + \alpha_2 = 0$. (Specifically, P_{i-1} generates $\alpha_{i-1} = F_{k_{i-1}}(id) - f(id)$ and P_{i+1} generates $\alpha_{i+1} = f(id) - F_{k_i}(id)$. Thus, $\alpha_{i-1} + \alpha_{i+1} = F_{k_{i-1}}(id) - f(id) + f(id) - F_{k_i}(id) = F_{k_{i-1}}(id) - F_{k_i}(id) = -\alpha_i$, as required.) The full proof follows via a straightforward reduction. ■

3.4.4 Wrapping Up

In the previous sections, we have proven that Protocol 3.3.3 computes any 3-ary functionality with perfect security in the F_{mult} -hybrid model, and that Protocol 7 computes the F_{mult} functionality with perfect security in the \mathcal{F}_{CR} -hybrid model. Finally, we have proved that Protocol 8 computes \mathcal{F}_{CR} with computational security (in the plain model) under the assumption that pseudorandom functions exist. (All of the above holds for a single corrupted party in the semi-honest model.) Using the fact that all our protocols are UC secure from [78] and thus applying the UC composition theorem of [29], we conclude with the following theorem:

Theorem 3.4.7. *Assume that F is a pseudorandom function, and let f be a 3-ary*

functionality. Then, Protocol 3.3.3 computes f with computational security, in the presence of one semi-honest corrupted party.

3.5 Security against Malicious Adversaries in the Client-Server Model

In this section, we consider the “client-server” model where the parties running the multi-party computation protocol are servers who receive the input shares of multiple clients and compute the output for them. This is the model used by Cybernetica in their Sharemind product [17]. In this model, the servers do not see any of the inputs nor any of the outputs. Rather, they receive *shares* of the inputs and send the clients shares of their output. Since the parties running the multi-party protocol do not have any input or output, it is possible to formulate an indistinguishability-based definition of security, saying that a corrupted server learns nothing. In this section, we present such a definition, and we prove that our protocol fulfills this definition of privacy even in the presence of a *malicious corrupted party*. We believe that this formalization is of independent interest, and could be used to make similar claims regarding other information-theoretic protocols like [14] and [17, 18]; namely, that although they are only secure in the presence of semi-honest adversaries, they are in fact *private in the presence of malicious adversaries*.

Before proceeding, we stress that a definition of privacy is strictly weaker than standard definitions of security for malicious adversaries. Most notably, *correctness* is not guaranteed and a malicious server may tamper with the output. In settings where the adversary may receive some feedback about the output, this may also reveal information about the input. Thus, our claim of privacy is only with respect to a malicious server who receives no information about the output.

We now prove that Protocol 3.3.3 fulfills Definition 2.4.3, when making the appropriate changes to the input (converting vectors of length N into 3-way additive shares for the parties running Protocol 3.3.3).

Theorem 3.5.1. *Let $f : ((\mathbb{Z}_{2^n})^*) \rightarrow ((\mathbb{Z}_{2^n})^*)$ be an N -party functionality and define the 3-party functionality g_f to be the function that receives 3 length- N input vectors that constitute additive-shares of the input vector \vec{v} to f and outputs 3 length- N vectors that constitute additive-shares of $f(\vec{v})$. If F is a pseudorandom function, then*

Protocol 3.3.3 applied to function g_f 1-securely computes f in the client-server model in the presence of malicious adversaries.

PROOF SKETCH: Correctness is also required for the semi-honest setting and this is therefore already implied by Theorem 3.4.1. In order to prove privacy, we need to show that the view of a malicious \mathcal{A} controlling one party when the input is \vec{v} is indistinguishable from its view when the input is \vec{v}' . We first prove that the views are *identical* when information-theoretic correlated randomness is used (as described in the beginning of Section 3.3.2).

First, intuitively, the views are identical with information-theoretic correlated randomness since all the adversary sees in every rounds is a random share. In order to see that this holds even when \mathcal{A} is *malicious*, observe that each share sent to the adversary is masked by a new value obtained from the correlated randomness. Thus, irrespective of what \mathcal{A} sends in every round, the value that it receives is a *random element*. Thus, its view is actually independent of the values that it sends.

Second, consider the view when Protocol 3.8 is used for computing \mathcal{F}_{CR} . In the setup phase, \mathcal{A} sends some value k_i and receives k_{i-1} . However, the security of the protocol is proven based on the pseudorandomness of the function keyed by k_{i+1} that \mathcal{A} does not see. Importantly to this case of malicious adversaries, k_{i+1} is chosen independently of what \mathcal{A} sends. Furthermore, the parties generate randomness from this point on using local computation only. Thus, the values generated by the honest parties are pseudorandom, irrespective of what \mathcal{A} sent. More formally, consider a reduction where $F_{k_{i+1}}$ is replaced by a truly random function f . Then, P_{i-1} computes $\alpha_{i-1} = F_{k_{i-1}}(id) - f(id)$ and P_{i+1} computes $\alpha_{i+1} = f(id) - F_{k_i}(id)$. Since k_i and k_{i-1} are fixed and independent of f , it follows that $\alpha_{i-1}, \alpha_{i+1}$ are random under the constraint that $\alpha_{i-1} + \alpha_{i+1} = -(F_{k_i}(id) - F_{k_{i-1}}(id)) = -\alpha_i$, as required. As we have stated, this holds irrespective of what value k_i that \mathcal{A} sent, and \mathcal{A} cannot influence the $\alpha_{i-1}, \alpha_{i+1}$ values computed since they involve local computation by the honest parties alone. Thus, the view in this case is indistinguishable from the view when the parties use information-theoretic correlated randomness. ■

3.6 Experimental Evaluation

3.6.1 Implementation Aspects

We implemented the protocol for Boolean circuits in C++ using standard optimizations known for multi-party computation. One specific optimization that we found to be of great importance was the use of Intel intrinsics for bit slicing operations; we describe this in more detail here. Since our protocol is extremely simple, running a single computation is very wasteful both with respect to CPU and network utilization. A significant portion of this waste is due to the fact that our protocol processes single bits only, whereas modern processors work on larger objects. We ran our protocol on 12800 operations in parallel by batching 128 operations together and running 100 of these in parallel. This batching works by *bit-slicing*.

Bit-Slicing The i th bit of input in 128 different inputs are sliced into a single string of length 128 (for each i). Likewise, the batched output bits need to be de-sliced into 128 separate outputs. This is a type of “matrix transpose” – see Figure 3.1 – and turns out to be very expensive. Indeed, a straightforward implementation of this bit slicing and de-slicing turned out to greatly dominate the overall execution time. Hence, we implemented fast bit-slicing and bit-deslicing methods using Intel SIMD intrinsics in order to reduce this cost.

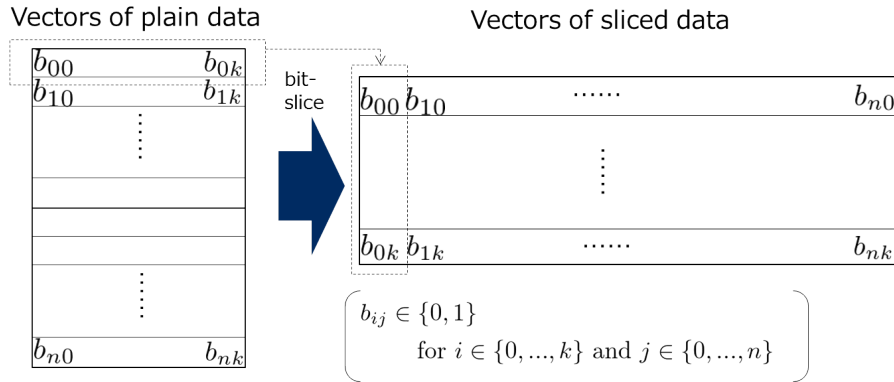


Figure 3.1: Bit-slice operation

The unit of our bit-slicing is 16 messages of length 8 bytes each (overall 128 bytes).

Thus, we start with:

$$\begin{aligned}
 m_0 &= (m_{0,0}, m_{0,1}, m_{0,2}, m_{0,3}, m_{0,4}, m_{0,5}, m_{0,6}, m_{0,7}) \\
 m_1 &= (m_{1,0}, m_{1,1}, m_{1,2}, m_{1,3}, m_{1,4}, m_{1,5}, m_{1,6}, m_{1,7}) \\
 &\dots \\
 m_{15} &= (m_{15,0}, m_{15,1}, m_{15,2}, m_{15,3}, m_{15,4}, m_{15,5}, m_{15,6}, m_{15,7}).
 \end{aligned}$$

Then, we apply the Intel intrinsics “unpack” instruction 32 times to obtain 8 messages, each of length 16 bytes:

$$\begin{aligned}
 m'_0 &= (m_{0,0}, m_{1,0}, \dots, m_{15,0}) \\
 m'_1 &= (m_{0,1}, m_{1,1}, \dots, m_{15,1}) \\
 &\dots \\
 m'_7 &= (m_{0,7}, m_{1,7}, \dots, m_{15,7}).
 \end{aligned}$$

The unpack instruction treats the 128 bit register as 16 single-byte values (8 low and 8 high), and has instructions to interleave either the low or the high bytes. This process is actually **byte-slicing** (since the “transpose”-type operation is carried out at the byte level and not the bit level). See Figure 3.2 for a graphic description of this operation.

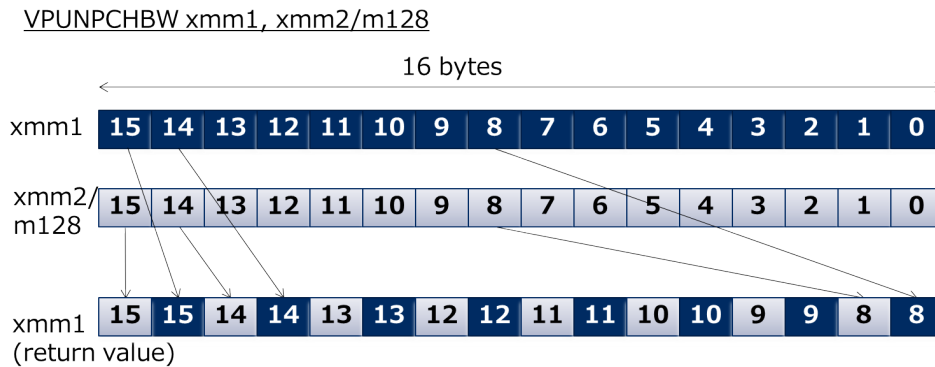


Figure 3.2: Unpack operation of AVX instruction set

The next step is to further slice the messages to the bit level. We do this applying the Intel `movmskb` 64 times to obtain the bit-sliced inputs. This instruction creates a 16-bit mask from the most significant bits of 16 signed or unsigned 8-bit integers in

a register and zeroes the upper bits. Thus, we are able to take the MSB of 16 bytes in a register in a *single cycle*, which is very fast. The `movmskb` instruction is depicted in Fig. 3.3.

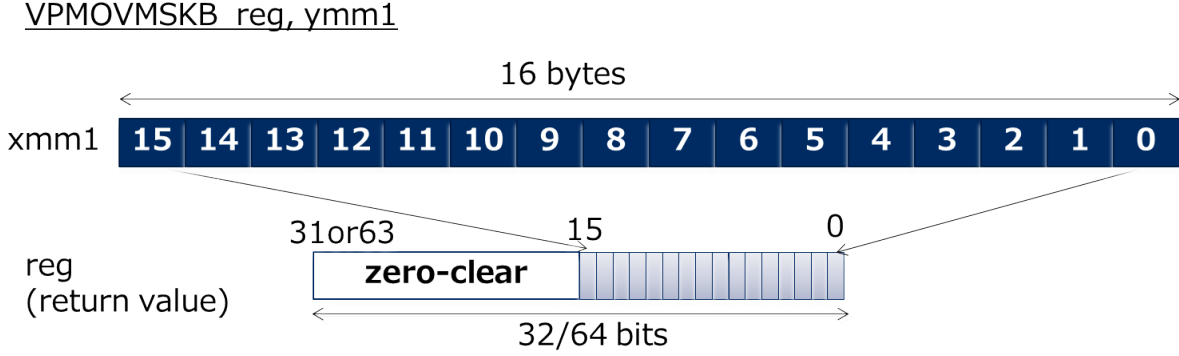


Figure 3.3: Moving masked bit operation of AVX instruction set

We apply the `movmskb` operation to each m'_i from the first step (note that each m'_i consists of 16 bytes, exactly as needed for `movmskb`). These optimizations were crucial for obtaining the high performance reported in this paper.

3.6.2 Result (1): Fast AES

We ran our implementation on a cluster of three mid-level servers connected by a 10Gbps LAN with a ping time of 0.13 ms. Each server has two Intel Xeon E5-2650 v3 2.3GHz CPUs with a total of 20 cores. We ran the implementation utilizing a different number of cores, from 1 through to 20. Each core was given 12800 computations which were carried out in parallel. (Since Intel intrinsics works on 128-bit registers, this means that inputs were sliced together in groups of 128 and then 100 of these were run in parallel by *each core*.) These computations can be with different keys since each MPC can have different inputs; this will be used in Section 3.6.3.

Observe that up to 10 cores, the throughput is stable at approximately 100,000 AES/sec per core. However, beyond 10 cores this begins to deteriorate. This is due to queuing between the kernel and the Network Interface Card (NIC). Specifically, when a single process utilizing a single CPU is used, that process has full control over the NIC. However, when multiple processes are run, utilizing high bandwidth, requests from each process are handled in a queue between the kernel and the NIC. This queuing increases network latency, and as each process spends more time waiting

for communication, CPU usage drops by a noticeable percentage. It is possible to overcome this by bypassing the kernel layer and communicating directly with the NIC. One approach for achieving this appeared in [107].

We ran each experiment 5 times; this was sufficient due to the very low variance as can be seen in Table 3.2. The results represent a 95% confidence interval.

Table 3.2: Experiment results running AES-CTR. The CPU column shows the average CPU utilization per core, and the network column is in Gbps per server. Latency is given in milliseconds.

Cores	AES/sec	Latency	CPU	Network
1	100,103 \pm 1632	128.5 \pm 2.1	73.3%	0.572
5	530,408 \pm 7219	121.2 \pm 1.7	62.2%	2.99
10	975,237 \pm 3049	131.9 \pm 0.4	54.0%	5.47
16	1,242,310 \pm 4154	165.7 \pm 0.4	49.5%	6.95
20	1,324,117 \pm 3721	194.2 \pm 0.9	49.6%	7.38

Recall that each core processed 12800 AES computations in parallel, and observe that with a latency of 129ms approximately 7 calls can be processed per second by each core. Thus, the approximate 100,000 AES computations per core per second are achieved in this way.

See Figures 3.4 and 3.5 for graphs showing the behavior of the implementation as higher throughputs are achieved.

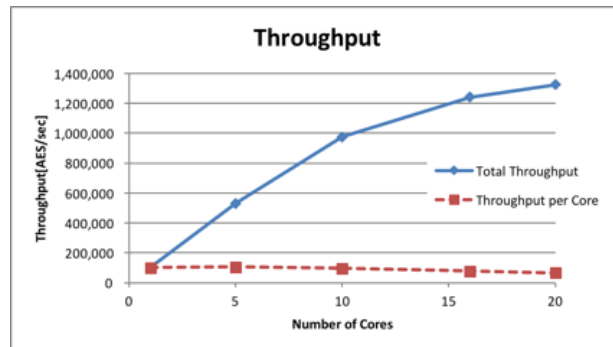


Figure 3.4: Throughput per core (AES computations)

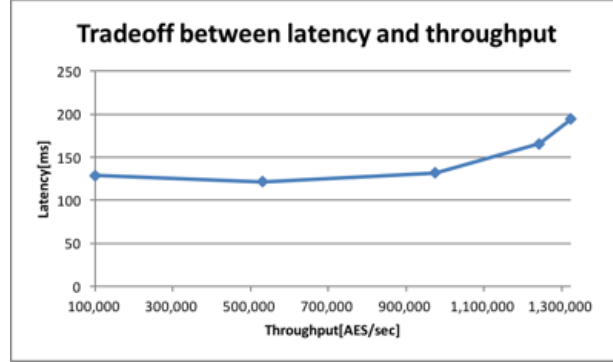


Figure 3.5: Latency versus throughput (AES)

Microbenchmarking. We measured the time spent on each part of the protocol, with the following results.

Protocol part	Percentage
Server bitslice and deslice	8.70%
AND and XOR gate computation	49.82%
Randomness generation	9.54%
Comm. delays between MPC servers	27.87%
Communication delays for input/output	4.07%

We remark that the long communication delays are due to the fact that the communication topology of our implementation is a ring. Thus, each party waits for two other messages to be processed before it receives its next message. In order to reduce this waste, the randomness generation is run during this delay. Thus, if the randomness generation was “free”, the communication delay would increase to 37.41% and it would not be any faster. This demonstrates that the efficiency improvements could be achieved by communicating in every step.

3.6.3 Result (2): Kerberos KDC with Shared Passwords

In order to demonstrate the potential of our protocol, we incorporate it into a real application. Kerberos is used for user authentication in many systems, most notably it is used by all Windows systems since Windows 2000. Kerberos uses the hashed user password as a key to encrypt a Ticket-Granting-Ticket (TGT) which contains a high-entropy cryptographic key which is used for all communications after the user logs in.

In Kerberos, a server breach is particularly devastating since the hashed password is all that is needed for impersonating a user. This is because the TGT is encrypted with the hashed user password and sent to the user. Thus, an attacker knowing the hashed password alone can decrypt the TGT. Microsoft’s Active Directory has suffered breaches in the past, and such a breach enables an attacker to impersonate every user in the organization.

In order to mitigate this risk, we consider a system where the hashed user passwords are XOR-shared between two servers (with different administrators), and secure multiparty computation is used to carry out the login authentication without ever reconstructing the hashed password. This makes it harder for an attacker to steal hashed passwords (needing to breach both servers) and also mitigates insider threats since no single administrator has access to the hashed user passwords. Since the ticket-granting-server’s long-term key is also very sensitive, this is also protected in the same way. The architecture of the Kerberos solution is depicted in Figure 3.6.

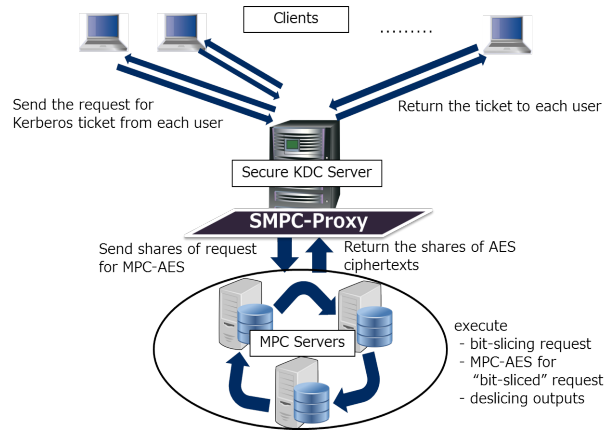


Figure 3.6: The Kerberos authentication using MPC

We took the Open Source MIT Kerberos and modified the encryption mode used to encrypt the TGT to counter mode. This is important since CBC mode does not enable parallel encryption and this would slow the encryption down significantly. In more detail, the authentication process in Kerberos has the following steps:

1. *Pre-authentication:* We use the PA-ENC-TIMESTAMP method, which means that the user encrypts the date using his hashed password as the key. This is a single AES block (and so ECB is used).

2. *TGT encryption*: A session key to be used by the user and ticket-granting server (TGS) to communicate later is generated. Then, the TGT (containing the client information and the session key) is generated and encrypted under the long-term key of the TGS. The TGT is 15 blocks of AES.
3. *Session-key and TGT encryption*: The session key and TGS are AES-encrypted with the user's hashed password.

Overall, the number of encryption blocks for a single user authentication is 33: one block for pre-authentication, 15 blocks for TGT encryption under the long-term key of the TGS, and 17 blocks for session-key and TGT encryption under the user key (this last encryption is 17 blocks due to the addition of the session key and header information).

In all of the above encryptions, when using the Kerberos encryption type `aes128-cts-hmac-sha1-96`, all of the encryption above is without HMAC authentication. (HMAC is only used for communication following these initial steps.) As we have mentioned, we implemented a Kerberos extension that uses counter mode instead of CBC (`cts` is CBC mode with ciphertext stealing). This is important for two reasons. First, CBC encryption cannot be parallelized and so each block must be encrypted after the previous block has been encrypted. In addition, the TGT cannot be encrypted under the user key until it has been encrypted under the long-term key of the TGS. However, when using counter mode, all of the AES computations can be carried out *in parallel*. Specifically, upon receiving a user authentication request together with a pre-authentication ciphertext, the following is carried out:

1. The servers running the secure computation protocol load the shares of the long-term key of the TGS and the shares of the user's key i.e., hash of the user's password).
2. Two random counters ctr_1 and ctr_2 are chosen.
3. 33 AES computations are run in parallel: a single AES decryption of the pre-authentication ciphertext, 15 AES encryptions of $ctr_1 + 1, \dots, ctr_1 + 15$, and 17 AES encryptions of $ctr_2 + 1, \dots, ctr_2 + 16$.
4. The preauthentication value is verified; if it is valid, then the server proceeds to the next step.

5. The output of the 15 AES encryptions using ctr_1 is XORed with the TGT.
6. The encrypted TGT from the previous step is concatenated with the session key and some header information. This is treated as a plaintext and XORed with the result of the 17 AES encryptions using ctr_2 .
7. The result of the previous step along with ctr_1 and ctr_2 is sent to the user.

This flow enables all of the AES computations to be carried out in parallel, yielding a latency of approximately 120 milliseconds. We remark that in order for the server to be able to process requests in bulk, a new set of AES encryptions is begun every 100 milliseconds. Thus, authentication requests are queued for at most 100 milliseconds (and on average 50ms) and then processed. This ensures that the overall latency (of a client) of processing an authentication request is approximately 200 milliseconds. This is a very reasonable time for an application like Kerberos where a user is involved in the authentication process.

Experimental results. In order to test our implementation, we ran the complete Kerberos login using the aforementioned cluster of three servers computing AES. The number of logins per second with a single core was 2,970, with 10 cores was 28,723 and with 16 cores was 36,521. Thus, our Kerberos implementation (that incorporates the extension described above in MIT-Kerberos) is able to support a significant login storm of over 40,000 user logins per second. This is sufficient even for very large organizations (if more is needed, then this can be achieved by simply using two clusters instead of one). Beyond the number of logins per second, it is important to ensure that the latency is low; otherwise, users will have to wait too long at login. This is the reason that we designed the TGT-generation process in a way that enables full parallelism of the AES operations. Our results give an average latency of the AES encryption via MPC at 110ms, and an average latency at the client (over a LAN) of 232ms. The increased time in the client is due to additional work carried out both by the client and the KDC, and due to the fact that requests are processed every 100ms.

Chapter 4 Foundation (2): Maliciously Secure 3-Party Computation based on Replicated Secret Sharing

4.1 Introduction

In this chapter, we present a high-throughput protocol for *three-party secure computation with an honest majority and security for malicious adversaries*. We optimize and implement the protocol of [56], that builds on the protocol of [6] that achieves a rate of over 7 billion AND gates per second, but with security only for *semi-honest* adversaries. The multiplication (AND gate) protocol of [6] is very simple; each party sends only a single bit to one other party and needs to compute only a few very simple AND and XOR operations. Security in the presence of malicious adversaries is achieved in [56] by using the cut-and-choose technique in order to generate many valid multiplication triples (shares of *secret* bits (a, b, c) where a, b are random and $c = ab$). These triples are then used to guarantee secure computation, as shown in [10]. This paradigm has been utilized in many protocols; see [79, 43, 70] for just a few examples.

The cut-and-choose method works by first generating many triples, but with the property that a malicious party can make $c \neq ab$. Then, some of the triples are fully “opened” and inspected, to verify that indeed $c = ab$. The rest of the triples are then grouped together in “buckets”; in each bucket, one triple is verified by using all the others in the bucket. This procedure has the property that the verified triple is valid (and a, b, c unknown), unless the unfortunate event occurs that *all* triples in the bucket are invalid. This method is effective since if the adversary causes many triples to be invalid then it is caught when opening triples, and if it makes only a few triples invalid then the chance of a bucket being “fully bad” is very small. The parameters needed (how many triples to open and how many in a bucket) are better – yielding

higher efficiency – as the number of triples generated overall increases. Since [6] is so efficient, it is possible to generate a very large number of triples very quickly and thereby obtain a very small bucket size. Using this idea, with a statistical security level of 2^{-40} , the protocol of [56] can generate 2^{20} triples while opening very few and using a bucket size of only 3. In the resulting protocol, each party sends only 10 bits per AND gate, providing the potential of achieving very high throughput.

We carried out a highly-optimized implementation of [56] and obtained a very impressive rate of approximately 500 million AND gates per second. However, our aim is to obtain even higher rates, and the microbenchmarking of our implementation pointed to some significant bottlenecks that must be overcome in order to achieve this. First, in order for cut-and-choose to work, the multiplication triples must be randomly divided into buckets. This requires permuting very large arrays, which turns out to be very expensive computationally due to the large number of cache misses involved (no cache-aware methods for random permutation are known and thus many cache misses occur). In order to understand the effect of this, note that on Intel Haswell chips the L1 cache latency is 4 cycles while the L3 cache latency is 42 cycles [1]. Thus, on a 3.4 GHz processor, the shuffling alone of one billion items in L3 cache would cost 11.7 seconds, making it impossible to achieve a rate of 1 billion gates per second (even using 20 cores). In contrast, in L1 cache the cost would be reduced to just 1.17 seconds, which when spread over 20 cores is not significant. Of course, this is a simplistic and inexact analysis; nevertheless, our experiments confirm this type of behavior.

In addition to addressing this problem, we design protocol variants of the protocol of [56] that require less communications. This is motivated by the assumption that bandwidth is a major factor in the efficiency of the protocol.

Protocol-design contributions. We optimized the protocol of [56], both improving its *theoretical efficiency* (e.g., communication) as well as its *practical efficiency* (e.g., via cache-aware design). We have the following contributions:

1. *Cache-efficient shuffling (Section 4.4.1):* We devise a cache-efficient method of permuting arrays that is sufficient for cut-and-choose. We stress that our method does *not* yield a truly random permutation of the items. Nevertheless, we provide a full combinatorial analysis proving that it suffices for the goal of cut-and-choose. We prove that the probability that an adversary can successfully cheat with our

new shuffle technique is the same as when carrying out a truly random permutation.

2. *Reduced bucket size (Section 4.4.2)*: As we have described above, in the protocol of [56], each party sends 10 bits to one other party for every AND gate (when computing 2^{20} AND gates and with a statistical security level of 2^{-40}). This is achieved by taking a bucket size of 3. We reduce the bucket size by 1 and thus the number of multiplication triples that need to be generated and used for verification by *one third*. This saves both communication and computation, and results in a concrete cost of each party sending 7 bits to one other party for every AND gate (instead of 10).
3. *On-demand with smaller buckets (Section 4.4.3)*: As will be described below, the improved protocol with smaller buckets works by running an additional shuffle on the array of multiplication triples after the actual circuit multiplications (AND gates) are computed. This is very problematic from a practical standpoint since many computations require far less than 2^{20} AND gates, and reshuffling the entire large array after every small computation is very wasteful. We therefore provide an additional protocol variant that achieves the same efficiency but without this limitation.

All of our protocol improvements and variants involve analyzing different combinatorial games that model what the adversary must do in order to successfully cheat. Since the parameters used in the protocol are crucial to efficiency, we provide (close to) tight analyses of all games.

Implementation contributions. We provide a high-quality implementation of the protocol of [56] and of our protocol variants. By profiling the code, we discovered that the SHA256 hash function computations specified in [56] take a considerable percentage of the computation time. We therefore show how to replace the use of a collision-resistant hash function with a secure MAC and preserve security; surprisingly, this alone resulted in approximately a 15% improvement in throughput. This is described in Section 4.4.4.

We implemented the different protocol variants and ran them on a cluster of three mid-level servers (2.3GHz CPUs with twenty cores) connected by a 10Gbps network. As we describe in Section 4.6, we used Intel vectorization and a series of optimizations

to achieve our results. Due to the practical limitations of the first variant with smaller buckets, we only implemented the on-demand version. The highlights are presented in Table 4.1. Observe that our fastest variant achieves a rate of over **1.1 billion AND-gates per second**, meaning that large scale secure computation *is* possible even for malicious adversaries.

Table 4.1: Implementation results; throughput

Protocol Variant	AND gates/sec	%CPU	Gbps
Baseline [56]	503,766,615	71.7%	4.55
Cache-efficient (SHA256)	765,448,459	64.84%	7.28
Smaller buckets, on-demand (SHA256)	988,216,830	65.8%	6.84
Smaller buckets, on-demand (MAC)	1,152,751,967	71.28%	7.89

Observe that the cache-efficient shuffle alone results in a *50% increase* in throughput, and our best protocol version is *2.3 times faster* than the protocol described in [56].

Offline/online. Our protocols can run in offline/online mode, where multiplication triples are generated in the offline phase and used to validate multiplications in the online phase. The protocol variants with smaller bucket size (items (2) and (3) above) both require additional work in the online phase to randomly match triples to gates. Thus, although these variants have higher throughput, they have a slightly slower online time (providing an interesting tradeoff). We measured the online time only of the fastest online version; this version achieves a processing rate of **2.1 billion AND gates per second** (using triples that were previously prepared in the offline phase).

Combinatorial analyses. As we have mentioned above, the combinatorial analyses used to prove the security of our different protocols are crucial for efficiency. Due to this observation, we prove some *independent* claims in Section 4.5 that are relevant to all cut-and-choose protocols. First, we ask the question as to whether having different-sized buckets can improve the parameters (intuitively, this is the case since it seems harder for an adversary to fill a bucket with all-bad items if it doesn’t know the size of the bucket). We show that this cannot help “much” and it is best to take

buckets of all the same size or of two sizes B and $B + 1$ for some B . Furthermore, we show that it is possible to somewhat tune the cheating probability of the adversary. Specifically, if a bucket-size B taken does not give a low enough cheating probability then we show that instead of increasing the bucket size to $B + 1$ (which is expensive), it is possible lower the cheating probability moderately at less expense.

4.2 Related Work

As we have described above, a long series of work has been carried out on making secure computation efficient, both for semi-honest and malicious adversaries. Recent works like [111] provide very good times for the setting of two parties and malicious adversaries (achieving a rate of 26,000 AND gates per second). This is far from the rates we achieve here. However, we stress that they work in a much more difficult setting, where there is no honest majority.

To the best of our knowledge, the only highly-efficient implemented protocol for the case of three parties with an honest majority and (full simulation-based security) for malicious adversaries is that of [93], which follows the garbled-circuit approach. Their protocol achieves a processing rate of approximately 480,000 AND gates per second on a 1Gbps network with single-core machines. One could therefore extrapolate that on a setup like ours, their protocol could achieve rates of approximately 5,000,000 AND gates per second. Note that by [93, Table 3] a single AES circuit of 7200 AND gates requires sending 750KB, or 104 bytes (832 bits) per gate. Thus, on a 10Gbps network their protocol *cannot* process more than 12 million AND gates per second (even assuming 100% utilization of the network, which is typically not possible, and that computation is not a factor). Our protocol is therefore at least *two orders of magnitude* faster. We stress, however, that the *latency* of [93] is much lower than ours, which makes sense given that it follows the GC approach.

The VIFF framework also considers an honest majority and has an implementation [40]. The offline time alone for preparing 1000 multiplications is approximately 5 seconds.¹ Clearly, on modern hardware, this would be considerably faster, but only by 1-2 orders of magnitude.

¹This is for 4 parties with at most 1 corrupted. However, when considering security with abort as we do here, the protocol of [40] can be adapted to 3 parties with 1 corrupted with approximately the same cost.

4.3 The Baseline Protocol

4.3.1 An Informal Description

In [56], a three-party protocol for securely computing any functionality (represented as a Boolean circuit) with security in the presence of malicious adversaries and an honest majority was presented. The protocol is extremely efficient; for a statistical cheating probability of 2^{-40} the protocol requires each party to send only 10 bits per AND gate. In this section, we describe the protocol and how it works. Our description is somewhat abstract and omits details about what exact secret sharing scheme is used, how values are checked and so on. This is due to the fact that all the techniques in this paper are general and work for any instantiation guaranteeing the properties that we describe below.

Background – multiplication triples. The protocol follows the paradigm of generating shares of multiplication triples (also known as “Beaver triples”) $([a], [b], [c])$ where $a, b, c \in \{0, 1\}$ such that $c = ab$, and $[x]$ denotes a sharing of x . As we have mentioned, this paradigm was introduced by [10] and has been used extensively to achieve efficient secure computation [79, 43, 70].

One popular purpose for using the multiplication triples is to construct MPCs under *dishonest-majority*. Assume we have a multiplication triple $([a], [b], [c])$. When $[x], [y]$ are given and we want to perform multiplication to obtain $[z] = [x \cdot y]$, each party computes $[\sigma] = [x] - [a]$ and $[\rho] = [y] - [b]$ (namely, *masking* $[x]$ and $[y]$ by random shares) and opens σ and ρ . Then, each party can compute $[z] := \sigma \cdot \rho + \sigma \cdot [b] + \rho \cdot [a] + [c]$. The nice feature of this procedure is that we can easily introduce a MPC multiplication protocol from *any* LSS scheme (even if it cannot perform MPC multiplication by itself) if we can prepare multiplication triples in some way. This technique has been used in a number of schemes to date.

However, in this paper, we focus on other interesting properties to realize cheater detection: it is possible to efficiently validate if a triple is correct (i.e., if $c = ab$) by opening it, and it is possible to efficiently verify if a triple $([a], [b], [c])$ is correct *without opening* it by using another triple $([x], [y], [z])$. This latter check is such that if one triple is correct and the other is not, then the adversary is always caught. Furthermore, nothing is learned about the values a, b, c (but the triple $([x], [y], [z])$ has been “wasted” and cannot be used again).

Protocol description: The protocol of [56] works as shown in Protocol 4.1.

Protocol 4.1 : Computing a function f with Malicious Adversary

- **Inputs and Auxiliary Input:** Same as in Protocol 3.1.
 - **The protocol – offline phase:** Generate multiplication triples by calling Protocol 2; let \vec{d} be the output.
 - **The protocol – online phase:**
 1. *Step 1 – generate random multiplication triples:* In this step, the parties generate a large number of triples $([a_i], [b_i], [c_i])$ with the guarantee that $[a_i], [b_i]$ are random and all sharings are *valid* (meaning that the secret sharing values held by the honest parties are consistent and of a well-defined value). However, a malicious party can cause $c_i \neq a_i b_i$. In [56] this is achieved in two steps; first generate random sharings of $[a_i], [b_i]$ and then run the semi-honest multiplication protocol of [6] to compute $[c_i]$. This multiplication protocol has the property that the result is *always a valid sharing*, but an adversary can cause $c_i \neq a_i b_i$ and thus it isn't necessarily correct.
 2. *Step 2 – validate the multiplication triples:* In this step, the parties validate that the triples generated are valid (meaning that $c_i = a_i b_i$). This is achieved by opening a few of the triples completely to check that they are valid, and to group the rest in “buckets” in which some of the triples are used to validate the others. The validation has the property that all the triples in a bucket are used to validate the first triple, so that if that triple is bad then the adversary is caught cheating unless all the triples in the bucket are bad. The triples are *randomly shuffled* in order to divide them into buckets, and the bucket-size taken so that the probability that there exists a bucket with all-bad triples is negligible. We denote by N the number of triples that need to be generated (i.e., output from this stage), by C the number of triples opened initially, and by B the bucket size. Thus, in order to output N triples in this step, the parties generate $BN + C$ triples in the previous step.
-

-
3. *Step 3 – circuit computation:* In this step, the parties securely share their input bits, and then run the semi-honest protocol of [6] up to (but not including) the stage where outputs are revealed. We note that this protocol reveals nothing, and so as long as correctness is preserved, then full security is obtained.
 4. *Step 4 – validation of circuit computation:* As we described above, the multiplication protocol used in the circuit computation always yields a valid sharing but not necessarily of the correct result. In this step, each multiplication in the circuit is validated using a multiplication triple generated in Step 2. This uses the exact same procedure of validating “with opening”; as explained above, this reveals nothing about the values used in the circuit multiplication but ensures that the result is correct.
 5. *Step 5 – output:* If all of the verifications passed, the parties securely reconstruct the secret sharings on the output wires in order to obtain the output.

The checks of the multiplication triples requires the parties to send values and verify that they have the same view. In order to reduce the bandwidth (which is one of the main aims), in the protocol of [56] the parties compare their views only at the end before the output is revealed, by sending a collision-resistant hash of their view which is very short. (A similar idea of checking only at the end was used in [79, 43]). Note that Steps 1–2 can be run in a separate **offline phase**, reducing latency in the **online phase** of Steps 3–5.

Efficiency. The above protocol can be instantiated very efficiently. For example, sharings of random values can be generated non-interactively, the basic multiplication requires each party sending only a single bit, and verification of correctness of triples can be deferred to the end. Furthermore, since multiplication triples can be generated so efficiently, it is possible to generate a huge amount at once (e.g., 2^{20}) which significantly reduces the overall number of triples required. This is due to the combinatorial analysis of the cut-and-choose game. Concretely, it was shown in [56] that for a cheating probability of 2^{-40} , one can generate $N = 2^{20}$ triples using bucket-size $B = 3$ and opening only $C = 3$ triples. Thus, overall $3N + 3$ triples must be generated. The communication cost of generating each triple initially is a single bit, the cost of each validation (in Steps 2 and 4) is 2 bits, and the cost of multiplying in

Step 3 is again 1 bit. Thus, the overall communication per AND gate is just 10 bits per party (3 bits to generate 3 triples, 4 bits to validate the first using the second and third, 1 bit to multiply the actual gate, and 2 bits to validate the multiplication).

Shuffling and generating buckets. The shuffling of Step 2 in [56] works by simply generating a single array of $M = BN + C$ triples and randomly permuting the entire array. Then, the first C triples are opened, and then each bucket is generated by taking B consecutive triples in the array. In our baseline implementation, we modified this process. Specifically, we generate 1 array of length N , and $B - 1$ arrays of length $N + C$. The arrays of length $N + C$ are independently shuffled and the last C triples in each of these arrays is opened and checked. Finally, the i th bucket is generated by the taking the i th triple in each of the arrays (for $i = 1, \dots, N$). This is easier to implement, and will also be needed in our later optimizations. We remark that this is actually a different combinatorial process than the one described and analyzed in [56], and thus must be proven. In Section 4.4.1, we show that this makes almost no difference, and an error of 2^{-40} is achieved when setting $N = 2^{20}$, $B = 3$ and $C = 1$ (practically the same as [56]).

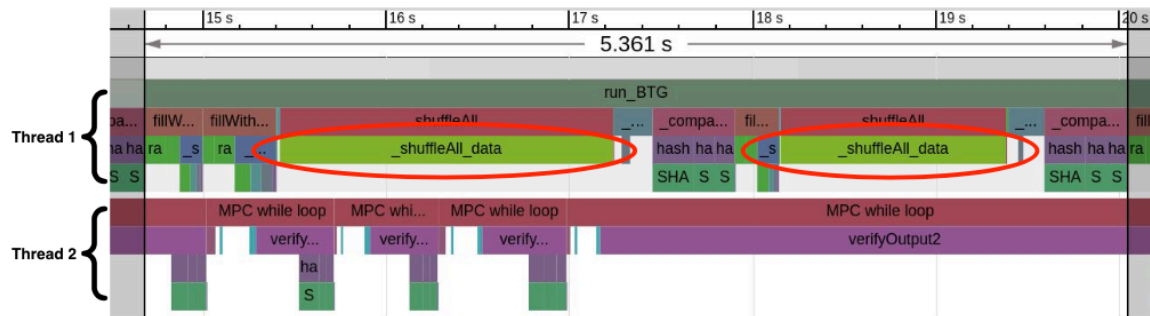


Figure 4.1: Microbenchmarking of the baseline implementation (the protocol of [56]), using the CxxProf C++ profiler

4.3.2 Implementation Results and Needed Optimizations

As we have discussed, the above protocol is highly efficient, requiring only 10 bits of communication per AND gate, and requiring only very simple operations. As such, one would expect that a good implementation could achieve a rate that is just a factor of 10 slower than the semi-honest protocol of [6] that computes 7.15 billion AND gates per second. However, our implementation yielded results which fall short

of this.

Specifically, on a cluster of three mid-level servers (Intel Xeon E5-2560 v3 2.3GHz with 20 cores) connected by a 10Gbps LAN with a ping time of 0.13 ms, our implementation of [56] achieves a rate of **503,766,615** AND gates per second. This is already very impressive for a protocol achieving malicious security. However, it is **14** times slower than the semi-honest protocol of [6], which is significantly more than the factor of 10 expected by a theoretical analysis.

In order to understand the cause of the inefficiency, see the microbenchmarking results in Figure 4.1. This is a slice showing one full execution of the protocol, with two threads: the first thread called `run_BTG` (*Beaver Triples Generator*) runs Steps 1–2 of the protocol to generate validated triples; these are then used in the second thread called `MPC while loop` to compute and validate the circuit computation (Steps 3–4 of the protocol). Our implementation works on blocks of 256-values at once (using the bit slicing described in [6]), and thus this demonstrates the generation and validation of 256 million triples and secure computation of the AES circuit approximately 47,000 times (utilizing 256 million AND gates).²

Observe that *over half the time* in `run_BTG` is spent on just randomly shuffling the arrays in Step 2 (dwarfing all other parts of the protocol). In hindsight, this makes sense since no cache-efficient random shuffle is known, and we use the best known method of Fisher-Yates [51]. Since we shuffle arrays of one million entries of size 256 bits each, this results in either main memory or L3 cache access at almost every swap (since L3 cache is shared between cores, it cannot be utilized when high throughput is targeted via the use of multiple cores). One attempt to solve this is to work with smaller arrays, and so a smaller N . However, in this case, a much larger bucket size will be needed in order to obtain a cheating bound of at most 2^{-40} , significantly harming performance.

Observe also that the fourth execution of `MPC while loop` of the second thread is extremely long. This is due to the fact that `MPC while loop` consumes triples generated by `run_BTG`. In this slice, the first three executions of `MPC while loop` use triples generated in previous executions of `run_BTG`, while the fourth execution of

²The *actual times* in the benchmark figure should be ignored since the benchmarking environment is on a local machine and not on the cluster.

MPC `while` loop is delayed until this `run_BTG` concludes. Thus, the circuit computation thread actually wastes approximately half its time waiting, making the entire system much less efficient.

4.4 Optimized Cheater Identification

In this section, we present multiple protocol improvements and optimizations to the protocol of [56]. Our variants are all focused on the combinatorial checks of the protocol, and thus do not require new simulation security proofs, but rather new bounds on the cheating probability of the adversary.

Our presentation throughout will assume subprotocols as described in Section 4.3.1: **(a)** generate random multiplication triples, **(b)** verify a triple “with opening”, **(c)** verify one triple using another “without opening”, and **(d)** verify semi-honest multiplication using a multiplication triple.

4.4.1 Cache-Efficient Shuffling for Cut-and-Choose

As we have discussed, one of the major bottlenecks of the protocol of [56] is the cost of random shuffling. In this section, we present a new shuffling process that is *cache efficient*. We stress that our method does not compute a true random permutation over the array. However, it does yield a permutation that is “random enough” for the purpose of cut-and-choose, meaning that the probability that an adversary can obtain a bucket with all bad triples is below the required error.

Informal description. The idea behind our shuffling method is to break the array into subarrays, internally shuffle each subarray separately, and then shuffle the subarrays themselves. By making each subarray small enough to fit into cache (L2 or possibly even L1), and by making the number of subarrays not too large, this yields a much more efficient shuffle. In more detail, recall that as described in Section 4.3.1, instead of shuffling one large array in the baseline protocol, we start with 1 subarray \vec{D}_1 of length N , and $B - 1$ subarrays $\vec{D}_2, \dots, \vec{D}_B$ each of size $N + C$, and we shuffle $\vec{D}_2, \dots, \vec{D}_B$. Our cache-efficient shuffling works by:

1. Splitting each array \vec{D}_k into L subarrays $\vec{D}_{k,1}, \dots, \vec{D}_{k,L}$.
2. Shuffling each subarray separately. (i.e., randomly permuting the entries inside each $\vec{D}_{k,i}$).

3. Shuffling the subarrays themselves.

This process is depicted in Figure 4.2.

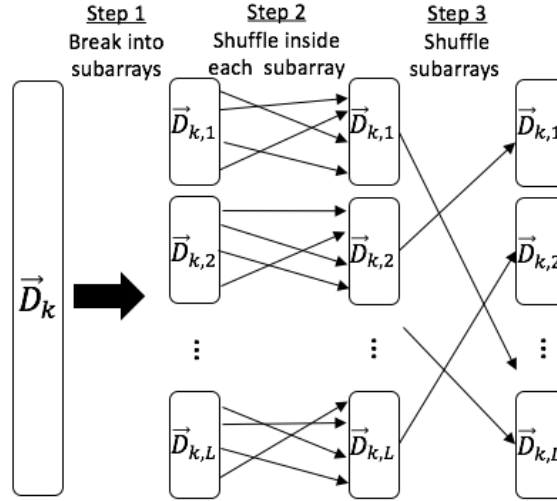


Figure 4.2: Cache-efficient shuffling method

We remark that in order to further improve efficiency, we do not shuffle the actual data but rather just the indices.³ This is much more efficient since it saves many memory copies; we elaborate on this further in Section 4.6.

As we will show, in order for this to be secure, it is necessary to open C triples in *each subarray*. Thus, $N/L + C$ triples are needed in each subarray, the size of each \vec{D}_k (for $k = 2, \dots, B$) is $L \cdot (N/L + C) = N + CL$, and the overall number of triples needed is $N + (B - 1)(N + CL)$. In addition, overall we execute a shuffling $(B - 1)(L + 1)$ times: $(B - 1)L$ times on the subarrays each of size $N/L + C$ and an additional $B - 1$ times on an array of size L . Interestingly, this means that the number of elements shuffled is slightly larger than previously; however, due to the memory efficiency, this is much faster. The formal description appears in Protocol 4.2.

³The protocol is highly efficient when using vectorization techniques, as described in Section 4.6. Thus, each item in the array is actual 256 triples and the data itself is 96 bytes.

Protocol 4.2 : Generating Valid Triples – Cache-Efficiently

- **Input:** The number N of triples to be generated.
- **Auxiliary input:** Parameters B, C, X, L , such that $N = (X - C)L$; N is the number of triples to be generated, B is the number of buckets, C the number of triples opened in each subarray, and $X = N/L + C$ is the size of each subarray.
- **The Protocol:**
 1. *Generate random sharings:* The parties generate $2M$ sharings of random values, for $M = 2(N + CL)(B - 1) + 2N$; denote the shares that they receive by $[[a_i], [b_i]]_{i=1}^{M/2}$.
 2. *Generate array \vec{D} of multiplication triples:* As in Step 1 of the informal description in Section 4.3.1.
 3. *Cut and bucket:* In this stage, the parties perform a first verification that the triples were generated correctly, by opening some of the triples.
 - (a) Each party splits \vec{D} into vectors $\vec{D}_1, \dots, \vec{D}_B$ such that \vec{D}_1 contains N triples and each \vec{D}_j for $j = 2, \dots, B$ contains $N + LC$ triples.
 - (b) For $k = 2$ to B : each party splits \vec{D}_k into L subarrays of equal size X , denoted by $\vec{D}_{k,1}, \dots, \vec{D}_{k,L}$.
 - (c) For $k = 2, \dots, B$ and $j = 1, \dots, L$: the parties jointly and securely generate a random permutation of the vector $\vec{D}_{k,j}$.
 - (d) For $k = 2, \dots, B$: the parties jointly and securely generate a random permutation of the vector $[1, \dots, L]$ and permute the subarrays in \vec{D}_k accordingly.
 - (e) For $k = 2, \dots, B$ and $j = 1, \dots, L$: The parties open and check each of the first C triples in $\vec{D}_{k,j}$, and remove them from $\vec{D}_{k,j}$. If a party rejects any check, it sends \perp to the other parties and outputs \perp .
 - (f) The remaining triples are divided into N sets of triples $\vec{B}_1, \dots, \vec{B}_N$, each of size B , such that the bucket \vec{B}_i contains the i 'th triple in $\vec{D}_1, \dots, \vec{D}_B$.
 4. *Check buckets:* In each bucket, $B - 1$ triples are used to validate the first (as in Step 2 of the informal description in Section 4.3.1).
- **Output:** The parties output \vec{d} .

Intuition – security. It is clear that our shuffling process does *not* generate a random permutation over the arrays $\vec{D}_2, \dots, \vec{D}_B$. However, for cut-and-choose to

work, it is seemingly necessary to truly randomly permute the arrays so that the adversary has the lowest probability possible of obtaining a bucket with all-bad triples. Despite this, we formally prove that our method does suffice; we first give some intuition.

Consider the simplistic case that the adversary generates one bad triple in each array \vec{D}_k . Then, for every k , the probability that after the shuffling a bad triple in \vec{D}_k will be located in the same index as the bad triple in \vec{D}_1 is $\frac{1}{N/L} \cdot \frac{1}{L} = \frac{1}{N}$ (after opening C triples, there are N/L in the subarray and L subarrays; the bad triples will match if they match inside their subarrays and the their subarrays are also matched). Observe that this probability is exactly the same as in the naive shuffling process where the entire array of N is shuffled in entirety.

A subtle issue that arises here is the need to open C triples in *each* of the subarrays $\vec{D}_{k,j}$. As we have mentioned, this means that the number of triples that need to be opened increases as L increases. We stress that this is necessary, and it does *not* suffice to open C triples only in the entire array. In order to see why, consider the following adversarial strategy: choose one subarray in *each* \vec{D}_k and make all the triples in the subarray bad. Then, the adversary wins if the no bad triple is opened (which happens with probability $1 - \frac{C}{N+C}$) and if the B bad subarray are permuted to the same position (which happens for each with probability $1/L$). The the overall probability that the adversary wins is close to $\frac{1}{LB}$ which is much too large (note that L is typically quite small). By opening balls in each $\vec{D}_{k,j}$, we prevent the adversary from corrupting an entire subarray (or many triples in a subarray).

Before proceeding, note that if we set $L = 1$, we obtain the basic shuffling of Section 4.3.1, and thus the combinatorial analysis provided next, applies to that case as well.

Proof of security – combinatorial analysis. We now prove that the adversary can cause the honest parties to output a bad triple in Protocol 2 with probability at most $\frac{1}{N^{B-1}}$. This bound is close to tight, and states that it suffices to take $B = 3$ for $N = 2^{20}$ exactly as proven in [56] for the baseline protocol. However, in contrast to the baseline protocol, here the parties must open $(B - 1)CL$ triples (instead of just $(B - 1)C$). Nevertheless, observe that the bound is actually *independent* of the choice of C and L . Thus, we can take $C = 1$ and we can take L to be whatever is suitable so that $N/L + C$ fits into the cache and L is not too large (if L is large then

many triples are wasted in opening and the permutation of $\{1, \dots, L\}$ would become expensive). Concretely, for $N = 2^{20}$ one could take $L = 512$ and then each subarray is of size 2049 (2048 plus one triple to be opened). Thus, $512(B - 1) = 1024$ triples overall are opened when generating 2^{20} triples, which is insignificant.

We start by defining a combinatorial game which is equivalent to the cut-and-bucket protocol using the optimized shuffling process. Recall that C denotes the number of triples that are opened in each subarray, B denotes the size of the bucket, L denotes the number of subarrays, and $X = N/L + C$ denotes the number of triples in each subarray.

Game₁(\mathcal{A}, X, L, B, C):

1. The adversary \mathcal{A} prepares a set D_1 of $(X - C)L$ balls and $B - 1$ sets D_2, \dots, D_B of $X \cdot L$ balls, such that each ball can be either **bad** or **good**.
2. Each set D_k is divided into L subsets $D_{k,1}, \dots, D_{k,L}$ of size X . Then, for each subset $D_{k,j}$ where $k \in \{2, \dots, B\}$ and $j \in [L]$, C balls are randomly chosen to be opened. If one of the opened balls is **bad** then output 0. Otherwise, the game proceeds to the next step.
3. Each subset $D_{k,j}$ where $k \in \{2, \dots, B\}$ and $j \in [L]$ is randomly permuted. Then, for each set D_k where $k \in \{2, \dots, B\}$, the subsets $D_{k,1}, \dots, D_{k,L}$ are randomly permuted inside D_k . Denote by $N = L(X - C)$ the size of each set after throwing the balls in the previous step. Then, the balls are divided into N buckets B_1, \dots, B_N , such that B_i contains the i th ball from each set D_k where $k \in [B]$.
4. The output of the game is 1 if and only if there exists i such that bucket B_i is **fully bad**, and all other buckets are either **fully bad** or **fully good**.

We begin by defining the **bad-ball profile** T_k of a set D_k to be the vector $(t_{k,1}, \dots, t_{k,L})$ where $t_{k,j}$ denotes the number of bad balls in the j 'th subarray of T_k . We say that two sets D_k, D_ℓ have **equivalent** bad-ball profiles if T_k is a permutation of T_ℓ (i.e., the vectors are comprised of exactly the same values, but possibly in a different order). We begin by proving that the adversary can only win if all sets have equivalent bad-ball profiles.

Lemma 4.4.1. *Let T_1, \dots, T_k be the bad-ball profiles of D_1, \dots, D_L . If $\text{Game}_1(\mathcal{A}, X, L, B, C) = 1$ then all the bad-ball profiles of T_1, \dots, T_k are equivalent.*

Proof. This is straightforward from the fact the adversary wins (and the output of the game is 1) only if for every $i \in [n]$ all the balls in the i th place of D_1, \dots, D_B are either bad or good. Formally, assume there exist k, ℓ such that T_k and T_ℓ are not equivalent. Then, for every permutation of the subsets in D_k and D_ℓ , there must exist some j such that $t_{k,j} \neq t_{\ell,j}$ after the permutation. Assume w.l.o.g that $t_{k,j} > t_{\ell,j}$. Then, for every possible permutation of the balls in $D_{k,j}$ and $D_{\ell,j}$, there must be a bad ball in $D_{k,j}$ that is placed in the same bucket as a good ball from $D_{\ell,j}$, and the adversary must lose. Thus, if the adversary wins, then all bad-ball profiles must be equivalent. ■

Next we prove that the best strategy for the adversary is to choose bad balls so that the same number of bad balls appear in every subset containing bad balls. Formally, we say that a bad-ball profile $T = (t_1, \dots, t_L)$ is **single-valued** if there exists a value t such for every $i = 1, \dots, \ell$ it holds that $t_i \in \{0, t\}$ (i.e., every subset has either zero or t bad balls). By Lemma 4.4.1 we know that all bad-ball profiles must be equivalent in order for the adversary to win. From here on, we can therefore assume that \mathcal{A} works in this way and there is a single bad-ball profile chosen by \mathcal{A} . Note that if the adversary chooses no bad balls then it cannot win. Thus, the bad-ball profile chosen by \mathcal{A} must have at least one non-zero value. The following lemma states that the adversary's winning probability is improved by choosing a single-valued bad-ball profile.

Lemma 4.4.2. *Let $T = (t_1, \dots, t_L)$ be the bad-ball profile chosen by \mathcal{A} and let t be a non-zero value in T . Let $T' = (t'_1, \dots, t'_L)$ be the bad-ball profile derived from T by setting $t'_i = t$ if $t_i \neq t$ and setting $t_i = 0$ otherwise (for every $i = 1, \dots, L$). Then, $\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \leq \Pr[\text{Game}_1(\mathcal{A}_{T'}, X, L, B, C) = 1]$, where $\mathcal{A}_{T'}$ chooses the balls exactly like \mathcal{A} except that it uses profile T' .*

Proof. Let T be the bad-ball profile chosen by \mathcal{A} and define T' as in the lemma. Let E_1 denote the event that no bad balls were detected when opening C balls in every subset, that all subsets containing t bad balls are matched together, and that all bad balls in these subsets containing t bad balls are matched in the same buckets. By

the definition of the game, it follows that $\Pr[\text{Game}_1(\mathcal{A}_{T'}, X, L, B, C) = 1] = \Pr[E_1]$. Next, define by E_2 the probability that in the game with \mathcal{A} , the subsets with a number of bad balls not equal to t are matched and bucketed together. Then,

$$\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] = \Pr[E_1 \wedge E_2].$$

We have that

$$\begin{aligned} \Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \\ &= \Pr[E_1 \wedge E_2] = \Pr[E_2 \mid E_1] \cdot \Pr[E_1] \\ &\leq \Pr[E_1] = \Pr[\text{Game}_1(\mathcal{A}_{T'}, X, L, B, C) = 1] \end{aligned}$$

and the lemma holds. \blacksquare

We are now ready to prove that the adversary can win in the game with probability at most $1/N^{B-1}$ (independently of C, N , as long as $C > 0$).

Theorem 4.4.3. *For every adversary \mathcal{A} , for every $L > 0$ and $0 < C < X$, it holds that*

$$\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \leq \frac{1}{N^{B-1}}$$

where $N = (X - C)L$.

Proof. By Lemma 4.4.2 it follows that the best strategy for the adversary is to choose some S subsets from each set, and to put exactly t bad balls in each of them, for some t , while all other subsets contain only good balls. Thus, overall there are SB subsets containing bad balls.

Next, we analyze the success probability of the adversary in the game when using this strategy. We define three *independent* events:

\mathbf{E}_c : the event that no bad balls were detected when opening C balls in each of the S sub-sets containing t bad balls in D_2, \dots, D_B . Since there are $\binom{X}{C}$ ways to choose C balls out of X balls, and $\binom{X-t}{C}$ ways to choose C balls without choosing any of the t bad balls, we obtain that

$$\Pr[E_c] = \left(\frac{\binom{X-t}{C}}{\binom{X}{C}} \right)^{S(B-1)} = \left(\frac{(X-t)!(X-C)!}{X!(X-t-C)!} \right)^{S(B-1)} \quad (5)$$

\mathbf{E}_L : the event that after permuting the subsets in D_2, \dots, D_B , the S subsets containing t bad balls are positioned at the same locations of the S subsets in D_1 . There

are $L!$ ways to permute the subsets in each D_k , and $S!(L-S)!$ ways to permute such that the subsets with t bad balls will be in the same location as in D_1 . Thus, we have

$$\Pr[E_L] = \left(\frac{S!(L-S)!}{L!} \right)^{B-1} = \binom{L}{S}^{-(B-1)}$$

\mathbf{E}_t : the event that after permuting the balls inside the subsets, all bad balls are positioned in the same location in D_1, \dots, D_B . For subset $D_{j,k}$ which contains t bad balls, there are $(X-C)!$ ways to permute it. In contrast, there are only $t!(X-C-t)!$ ways to permute it such that the bad balls will be in the same location of the bad balls in $D_{1,k}$. Since there are S subsets with t bad balls in each set, we have that

$$\Pr[E_t] = \left(\frac{t!(X-C-t)!}{(X-C)!} \right)^{S(B-1)}. \quad (6)$$

Combining the above three equations and noting that the product of Eq. (5) and Eq. (6) equals $\frac{t!(X-t)!}{X!} = \binom{X}{t}^{-1}$, we conclude that

$$\begin{aligned} \Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] &= \Pr[E_c \wedge E_L \wedge E_t] = \Pr[E_c] \cdot \Pr[E_L] \cdot \Pr[E_t] \\ &= \binom{L}{S}^{-(B-1)} \binom{X}{t}^{-S(B-1)}. \end{aligned} \quad (7)$$

Next, observe that for the adversary to win it must hold that $t \leq X-C < X$ and $S > 0$. Thus, we can use the fact that for every $0 < t < X$ it holds that $\binom{X}{t} \geq \binom{X}{1}$. In contrast, the adversary may choose to corrupt all subarrays. i.e., set $S = L$. Thus, we consider two cases.

- *Case 1: $S = L$.* In this case, we obtain that

$$\begin{aligned} \Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] &= \binom{L}{L}^{-(B-1)} \binom{X}{t}^{-L(B-1)} \\ &\leq \binom{X}{1}^{-L(B-1)} = \frac{1}{X^{L(B-1)}}. \end{aligned}$$

- *Case 2: $0 < S < L$.* In this case, we obtain that

$$\begin{aligned} \Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] &\leq \binom{L}{1}^{-(B-1)} \binom{X}{1}^{-S(B-1)} \\ &= \frac{1}{(L \cdot X^S)^{B-1}} \leq \frac{1}{(L \cdot X)^{B-1}} \end{aligned}$$

Since for every $L > 0$ and $X > 1$ (as assumed in the theorem) it holds that $L \cdot X \leq X^L$, we conclude that

$$\begin{aligned} & \Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \\ & \leq \max\left(\frac{1}{X^{L(B-1)}}, \frac{1}{(L \cdot X)^{B-1}}\right) = \frac{1}{(L \cdot X)^{B-1}} \\ & \leq \frac{1}{(L(X - C))^{B-1}} = \frac{1}{N^{B-1}} \end{aligned}$$

■

By setting $\frac{1}{N^{B-1}} \leq 2^{-\sigma}$ in Theorem 4.4.3, we conclude:

Corollary 4.4.4. *If L, X, C and B are chosen such that $\sigma \leq (B - 1) \log N$ where $L > 0$, $X > C > 0$ and $N = (X - C)L$, then for every adversary \mathcal{A} , it holds that $\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \leq 2^{-\sigma}$*

Concrete parameters. Observe that for $N = 2^{20}$, it suffices to set $B = 3$ and $C = 1$ and for any L we have that the adversary wins with probability at most 2^{-40} . This thus achieves the tight analysis provided in [56] when a *cache-inefficient* shuffle is used. In our implementation, we take $N = 2^{20}$ and $L = 2^9$; thus we have 512 subarrays of size 2048 each. Recall that we actually only shuffle the indices; for a subarray of length 2048 we need indices of size 2 bytes and so the entire subarray to be shuffled is 4096 bytes = 4KB. This fits into the L1 cache on most processors making the shuffle very fast.

4.4.2 Reducing Bucket-Size and Communication

Clearly, the major cost of the protocol is in generating, shuffling and checking the triples. If it were possible to reduce the size of the buckets needed, this would in turn reduce the number of triples to be generated and result in a considerable saving. In particular, the protocol of [56] uses a bucket size of 3 and requires that each party send 10 bits per AND gate; this places a strict lower bound on performance dependent on the available bandwidth. In this section, we show how to reduce the bucket size by 1 (concretely from 3 to 2) and thereby reduce the number of triples generated by 1/3, reducing computation and communication. Formally, we present an improvement that reduces the cheating probability of the adversary from $\frac{1}{N^{B-1}}$ to

$\frac{1}{N^B}$, thus enabling us to use $B' = B - 1$. Thus, if previously we needed to generate approximately 3 million triples in order to compute 1 million AND gates, in this section we show how the same level of security can be achieved using only *2 million* triples. Overall, this reduces communication from 10 bits per AND gate to 7 bits per AND gate (since 1 bit is needed to generate a triple and 2 bits are needed to verify each triple using another).

Protocol 4.3 : Computing f with Malicious Adversaries – Smaller Buckets

- **Inputs and Auxiliary Input:** Same as in Protocol 1; In addition, the parties hold a parameter L .
 - **The protocol – offline phase:** Generate multiplication triples by calling Protocol 2; let \vec{d} be the output.
 - **The protocol – online phase:**
 1. *Input sharing and circuit emulation:* Exactly as in Protocol 1.
 2. *Reshuffle stage:* The parties jointly and securely generate a random permutation over $\{1, \dots, N\}$ and then each locally shuffle \vec{d} accordingly.
 3. *Verification and output stages:* Exactly as in Protocol 1.
-

The idea behind the protocol improvement is as follows. The verification of a multiplication gate in the circuit uses one multiplication triple, with the property that if the gate is incorrect and the triple is valid (meaning that $c = ab$), then the adversary will be caught with probability 1. Thus, as long as all triples are valid with very high probability, the adversary cannot cheat. The improvement that we propose here works by observing that if a correct multiplication gate is verified using an incorrect triple *or* an incorrect multiplication gate is verified using a correct triple, then the adversary will be caught. Thus, if the array of multiplication triples is randomly shuffled *after* the circuit is computed, then the adversary can only successfully cheat if the random shuffle happens to match good triples with good gates and bad triples with bad gates. As we will see, this significantly reduces the probability that the

adversary can cheat and so the bucket size can be reduced by 1. Note that although the number of triples is reduced (since the bucket size is reduced), the number of shuffles remains the same.

Observe that in the reshuffle stage in Protocol 4.3, the random permutation is computed over the *entire* array, and does not use the cache-efficient shuffling of Section 4.4.1. This is due to the fact that unlike the triples generated in the preprocessing/offline phase, no triples of the circuit emulation phase can be opened. Thus, the adversary could actually make as many triples as it wishes in the circuit emulation phase be incorrect. In order to see why this is a problem, consider the case that the adversary choose to corrupt one subarray in each of $\vec{D}_1, \dots, \vec{D}_B$ and make $X - 1$ out of the X triples incorrect. Since $C = 1$, this implies that the adversary is *not caught* when opening triples in subarrays $\vec{D}_2, \dots, \vec{D}_B$ with probability X^{-B+1} . Furthermore, the probability that these subarrays with all-bad triples are matched equals L^{-B+1} . Thus, the adversary succeeds in have an all-bad bucket in the preprocessing phase with probability $\frac{1}{(XL)^{B-1}} < \frac{1}{N^{B-1}}$ as proven in Theorem 4.4.3. Now, if the cache-efficient shuffle is further used in the circuit computation phase, then the adversary can make a subarray all-bad there as well (recall that nothing is opened) and this will be matched with probability $1/L$ only. Thus, the overall cheating probability is bounded by $\frac{1}{L} \cdot \frac{1}{N^{B-1}} \gg \frac{1}{N^B}$. As a result, the shuffling procedure used in the online circuit-computation phase is a full permutation, and not the cache-efficient method of Section 4.4.1. We remark that even when using a full permutation shuffle, we need to make an additional assumption regarding the parameters. However, this assumption is fortunately very mild and easy to meet, as will be apparent below.

As previously, we begin by defining a combinatorial game to model this protocol variant.

Game₂(\mathcal{A}, X, L, B, C):

1. Run Game₁(\mathcal{A}, X, L, B, C) once. If the output is 0, then output 0. Otherwise, proceed to the next step with the buckets B_1, \dots, B_N .
2. The adversary \mathcal{A} prepares an additional set \vec{d} of N balls where each ball can be either **bad** or **good**.
3. The set \vec{d} is shuffled. Then, the i th ball in \vec{d} is added to the bucket B_i .

4. The output of the game is 1 if and only if each bucket is **fully good** or **fully bad**.

Note that in this game we do not explicitly require that the adversary can only win if there exists a bucket that is fully bad, since this condition is already fulfilled by the execution of Game_1 in the first step. We proceed to bound the winning probability of the adversary in this game.

Theorem 4.4.5. *If $B \geq 2$, then for every adversary \mathcal{A} and for every $L > 0$ and $0 < C < X$ such that $X^L \geq (X \cdot L)^2$, it holds that*

$$\Pr[\text{Game}_2(\mathcal{A}, X, L, B, C) = 1] \leq \frac{1}{N^B}$$

where $N = (X - C)L$.

Proof. Assume that the adversary chooses to corrupt exactly S subsets in Game_1 (the first step of Game_2) by inserts exactly t bad balls in each (recall that this strategy is always better, as proven in Lemma 4.4.2). Then, as shown in Eq. (7) in the proof of Theorem 4.4.3, it holds that

$$\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] = \binom{L}{S}^{-(B-1)} \binom{X}{t}^{-S(B-1)}.$$

Next, it is easy to see that for the adversary to win in Game_2 , it must choose exactly $S \cdot t$ bad balls in \vec{d} (otherwise a good and bad ball with certainly be in the same bucket). There are $\binom{N}{S \cdot t}$ ways of matching the $S \cdot t$ bad balls in \vec{d} , and there is exactly one way in which the adversary wins (this is where all $S \cdot t$ match the bad balls from Game_1). Thus, the probability that the adversary wins is

$$\begin{aligned} & \Pr[\text{Game}_2(\mathcal{A}, X, L, B, C) = 1] \\ &= \binom{N}{S \cdot t}^{-1} \cdot \Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1]. \end{aligned} \quad (8)$$

We separately consider two cases:

- *Case 1 – $S \cdot t < N$:* Applying Eq. (8) and the fact that $\binom{N}{S \cdot t}^{-1}$ is maximized for $S \cdot t = 1$ (since $S \cdot t$ cannot equal 0 or N)

$$\Pr[\text{Game}_2(\mathcal{A}, X, L, B, C) = 1] \leq \binom{N}{1}^{-1} \cdot \frac{1}{N^{B-1}} = \frac{1}{N^B}$$

- *Case 2* – $S \cdot t = N$: Observe that we cannot use Eq. (8) here since $\binom{N}{N} = 1$. We therefore prove the bound using Eq. (7). Here $S = L$ and so $\binom{L}{S} = 1$ and $t = X - C$ (note that $t < X$ since $C > 0$). Plugging this into Eq. (7) we have

$$\begin{aligned} & \Pr[\text{Game}_2(\mathcal{A}, X, L, B, C) = 1] \\ &= \binom{X}{X-C}^{-L(B-1)} \leq \binom{X}{1}^{-L(B-1)} = \frac{1}{X^{L(B-1)}}. \end{aligned}$$

Now, using the assumption that $X^L \geq (X \cdot L)^2$, which implies $X^{L(B-1)} \geq (X \cdot L)^{2(B-1)} \geq (X \cdot L)^B$ when $B \geq 2$ (which is indeed the minimal size of a bucket as assumed in the theorem), we obtain that

$$\begin{aligned} & \Pr[\text{Game}_2(\mathcal{A}, X, L, B, C) = 1] \\ &= \frac{1}{X^{L(B-1)}} \leq \frac{1}{(L \cdot X)^B} \leq \frac{1}{(L \cdot (X - C))^B} = \frac{1}{N^B}. \end{aligned}$$

■

We have the following corollary:

Corollary 4.4.6. *Let L, X, C and B be such that $\sigma \leq B \log N$ where $B \geq 2$, $L < 0$, $0 < C < X > 0$, $X^L \geq (X \cdot L)^2$ and $N = (X - C)L$. Then for every adversary \mathcal{A} , it holds that $\Pr[\text{Game}_2(\mathcal{A}, X, L, B, C) = 1] \leq 2^{-\sigma}$*

Concrete parameters and a tradeoff. As we have described above, this shows that setting $C = 1$, $B = 2$ and X, L such that $N = (X - C)L = 2^{20}$ yields a security bound of 2^{-40} as desired. Thus, we can reduce the size of each bucket by 1, and can use only 2 arrays in the triple generation phase (shuffling just one of them), at the expense of an additional shuffle in the online phase.

Clearly, in some cases one would not settle on any increase of the online work. Nevertheless, our analysis gives a clear trade-off of the offline communication complexity vs. the online computational complexity.

The latency vs throughput tradeoff. By reducing the number of triples sent and by reducing the communication, the protocol improvement here should considerably improve throughput. However, it is important to note that the fact that the online shuffle is not cache efficient means that the throughput increase is not optimal. In addition, it also means that the *online time* is considerably increased. Thus, when the

secure computation is used in an application where low latency is needed, then this improvement may not be suitable. However, when the focus is on high throughput secure computation, this is of importance.

Practical limitations. Although theoretically attractive, in most practical settings, the implementation of this protocol improvement is actually very problematic. Specifically, if a circuit computation involving N AND gates is used for a large N , then the improvement is suitable. However, in many (if not most) cases, circuits of smaller sizes are used and a large N is desired in order to achieve good parameters. For example, 2^{20} triples suffice for approximately 180 AES computation. In such a case, this protocol variant cannot be used. In particular, either the application has to wait for all AES computations to complete before beginning verification of the first (recall that the shuffle must take place *after* the circuit computation) or a full shuffle of what is left of the large array must be carried out after each computation. The former solution is completely unreasonable for most applications and the latter solution will result in a very significant performance penalty. We address this issue in the next section.

4.4.3 Smaller Buckets With On-Demand Secure Computation

In this section, we address the problem described at the end of Section 4.4.2. Specifically, we describe a protocol variant that has smaller buckets as in Section 4.4.2, but enables the utilization of multiplication triples *on demand* without reshuffling large arrays multiple times. Thus, this protocol variant is suitable for settings where many triples are generated and then used on-demand as requests for secure computations are received by an application.

The protocol variant that we present here, described in Protocol 4, works in the following way. First, we generate 2 arrays \vec{d}_1, \vec{d}_2 of N multiplication triples each, using Protocol 2 (and using a smaller B as in Section 4.4.2). Then, in order to verify a multiplication gate, a random triple is chosen from \vec{d}_1 and replaced with the next unused triple in \vec{d}_2 . After N multiplication gates have been processed, the triples in \vec{d}_2 will be all used and Protocol 2 will be called again to replenish it. Note that \vec{d}_1 always contains N triples, as any used triple is immediately replaced using \vec{d}_2 .

Protocol 4.4 : Computing f with Malicious Adversaries – On-Demand Shuffling and Smaller Buckets

- **Inputs and Auxiliary Input:** Same as in Protocol 1.
 - **The protocol – triple initialization:**
 1. The parties run Protocol 2 twice with input N and obtain two vectors \vec{d}_1, \vec{d}_2 of sharings of random multiplication triples.
 - **The protocol – circuit computation:** Upon receiving a request to compute a circuit:
 1. *Sharing the inputs:* Same as in Protocol 1.
 2. *Circuit emulation:* Same as in Protocol 1.
 3. *Verification stage:* Before the secrets on the output wires are reconstructed, the parties verify that all the multiplications were carried out correctly, as follows. For $k = 1, \dots, N$:
 - (a) Denote by $([x], [y])$ the shares of the input wires to the k th AND gate, and denote by $[z]$ the shares of the output wire of the k th AND gate.
 - (b) The parties run a secure coin-tossing protocol in order to generate a random $j \in [N]$. (In [56], it is shown that secure coin-tossing can be non-interactively and efficiently computed in this setting.)
 - (c) The parties check the triple $([x], [y], [z])$ using $([a_j], [b_j], [c_j])$ (the j th triple in \vec{d}_1).
 - (d) If a party rejects any of the checks, it sends \perp to the other parties and outputs \perp .
 - (e) Each party replaces its shares of $([a_j], [b_j], [c_j])$ in \vec{d}_1 with the next unused triple in \vec{d}_2 .
 4. *Output reconstruction and output:* Same as in Protocol 1.
 - **Replenish:** If \vec{d}_2 is empty (or close to empty) then the parties run Protocol 2 with input N to obtain a new \vec{d}_2 .
-

As before, we need to show that this way of working achieves the same level of security as when a full shuffle is run on the array. Formally, we will show that for N triples and buckets of size B , the probability that the adversary succeeds in cheating is bounded by $\frac{1}{N^B}$, just as in Section 4.4.2. Note that Protocol 4 as described is actually *continuous* and does not halt. Nevertheless, for simplicity, we present the bound for the case of computing N gates, and leave the continuous analysis to the full version.

In order to prove the bound, we begin by defining the combinatorial game $\text{Game}_3(\mathcal{A}, X, L, B, C)$ which is equivalent to the process described in Protocol 4.

$\text{Game}_3(\mathcal{A}, X, L, B, C)$:

1. Run steps 1-3 of $\text{Game}_1(\mathcal{A}, X, L, B, C)$ twice to receive two lists of buckets B_1, \dots, B_N and B'_1, \dots, B'_N .
2. If all buckets are either **fully good** or **fully bad** proceed to the next step. Otherwise, output 0.
3. The adversary \mathcal{A} prepares N new balls denoted by b_1, \dots, b_N , where each ball can be either **bad** or **good**, with the requirement that at least one of the balls must be bad.
4. For $i = 1$ to N :
 - (a) The ball b_i is thrown into a random bucket B_k ($k \in [N]$).
 - (b) If the bucket B_k is **fully bad** output 1.
 - (c) If the bucket B_k is not **fully good** or **fully bad** output 0.
 - (d) Replace B_k with the bucket B'_i .

Observe that in this game, the adversary is forced to choose a bad ball only when it prepares the N additional balls. This means that in order for it to win, there must be at least one bad bucket among B_1, \dots, B_N . For this to happen, the adversary must win in at least one of Game_1 executions. Thus, in the proof of the following theorem, we will use the bound stating that the probability that the adversary wins in Game_1 is at most $1/N^{B-1}$. In addition, note that from the condition in the last step, the adversary wins if and only if the *first bad ball* is thrown into a fully bad bucket (even

if a bad ball is later thrown into a fully good bucket meaning that the adversary will be detected). This is in contrast to previous games where the adversary only wins if *all* bad balls are thrown into fully bad buckets. This is due to the fact that output may be provided after only using some of the triples. If one of the triples was bad, then this will be a breach of security, and the fact that the adversary is caught later does not help (in the sense that security was already broken). Thus, cheating must be detected at the first bad ball and no later.

For the sake of simplicity and since this is what we use in our implementation, we concretely consider the case of $B = 2$ and our aim is to prove that the probability that the adversary cheats is at most $1/N^2$ (which equals 2^{-40} when $N = 2^{20}$). The proof of the general case will appear in the full version. In this game, unlike Sections 4.4.1 and 4.4.2, we actually need to open at least $C = 3$ triples in each subarray. We will explain why this is necessary at the end of the proof.

We prove the theorem under the assumptions that $X > L + C$ (meaning that the number of subarrays is less than the size of each subarray), that $L \geq 5$ (meaning that there are at least 5 subarrays), that $C \geq 3$, and that $X - C \geq 6$ (meaning that the subarrays are at least of size $C + 6$ which can equal 9). All of these conditions are fulfilled for reasonable choices of parameters in practice.

Theorem 4.4.7. *Let $B = 2$ and assume $X > L + C$. Then for every adversary \mathcal{A} and for every $L \geq 5$, $X - C \geq 6$ and $3 \leq C < X$ it holds that*

$$\Pr[\text{Game}_3(\mathcal{A}, X, L, B, C) = 1] \leq \frac{1}{N^2}$$

where $N = (X - C)L$.

Proof. In order to win the game, \mathcal{A} must choose bad balls in at least one of Game_1 executions. If \mathcal{A} chooses bad balls in both executions, then the theorem follows directly from Theorem 4.4.3, since \mathcal{A} wins in two executions of Game_1 with probability only $\frac{1}{N^{B-1}} \cdot \frac{1}{N^{B-1}} = \frac{1}{N^{2B-2}} \leq \frac{1}{N^B}$, where the last inequality holds when $B \geq 2$ as assumed, in the theorem.

Thus, for the remainder of the proof we assume that \mathcal{A} chose bad balls in exactly one of Game_1 executions only (note that the cases are mutually exclusive and so the probability of winning is the maximum probability of both cases).

Denote by S the number of subsets that contain bad buckets after the Game_1 executions (recall that we consider the case only that these are all in the same Game_1

execution), and let t be the number of bad buckets (in the proof of Theorem 4.4.3, note that t denotes the number of bad balls in the subarray; if the adversary is not caught then this is equivalent to the number of bad buckets). By Eq. (7) we have that

$$\begin{aligned} \Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] &= \binom{L}{S}^{-(B-1)} \binom{X}{t}^{-S(B-1)} \\ &= \binom{L}{S}^{-1} \binom{X}{t}^{-S} \end{aligned}$$

where the second equality follows since here we consider only the case of $B = 2$. We separately consider the cases that $S = 1$, $S = 2$, $S = 3$ and $S \geq 4$.

Case 1 – $S = 1$: In this case, we have

$$\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] = \binom{L}{1}^{-1} \binom{X}{t}^{-1} = \frac{1}{L} \cdot \binom{X}{t}^{-1}.$$

If $t = 1$, then $\binom{X}{t}^{-1} = \frac{1}{X} < \frac{L}{N}$ and so the probability that \mathcal{A} wins in Game_1 is at most $\frac{1}{N}$. In this case, in the latter steps in Game_3 , \mathcal{A} can only win by choosing exactly one bad ball out of b_1, \dots, b_N . Now, this ball is thrown into a random bucket, and there is at most one bad bucket (note that if the bad bucket is in the second array then depending on where the bad ball is, it may not even be possible for it to be chosen). Thus, the probability that it will be thrown into that bucket (which is essential for \mathcal{A} to win) is at most $\frac{1}{N}$. Overall, we have that \mathcal{A} can win Game_3 with probability at most $\frac{1}{N^2}$ (since \mathcal{A} must both win in Game_1 and have the bad ball thrown in the single bad bucket).

Next, if $t = 2$, we have that

$$\binom{X}{t}^{-1} = \binom{X}{2}^{-1} = \frac{2}{X(X-1)} < \frac{2}{(X-C)^2} = \frac{2L^2}{N^2}.$$

Thus,

$$\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] < \frac{2L}{N^2}.$$

Now, in the later phase of Game_3 , we have that there are at most 2 bad buckets out of N buckets overall.⁴ Thus, for each bad ball, the probability that it will be

⁴This holds since $t = 2$ and thus 2 bad buckets were generated in Game_1 . Note that there are *at most* 2 bad buckets at this stage and not necessarily 2 since the bad buckets in Game_1 may have been generated in the second set.

thrown into a bad bucket is at most $\frac{2}{N}$. Combining these together, we have that the adversary can win in Game_3 with probability at most

$$\frac{2L}{N^2} \cdot \frac{2}{N} = \frac{4L}{N^3} < \frac{1}{N^2}$$

where the inequality follows since we assume $X - C \geq 6 > 4$ and thus $4L < (X - C)L = N$.

Finally, if $t \geq 3$, we have that

$$\begin{aligned} \binom{X}{t}^{-1} &\leq \binom{X}{3}^{-1} = \frac{6}{X(X-1)(X-2)} \\ &< \frac{6}{(X-C)^3} = \frac{6L^3}{N^3}. \end{aligned}$$

We stress that $\binom{X}{t}^{-1} \leq \binom{X}{3}^{-1}$ is only true since we take $C \geq 3$, because otherwise $\binom{X}{t}^{-1}$ would be smaller for $t = X - 2$ or $t = X - 1$. However, when three balls are checked, if the adversary sets $t \geq X - 2$ it will certainly be caught (since at least one bad ball will always be checked). Thus,

$$\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] < \frac{6L^2}{N^3}.$$

Now, in the later phase of Game_3 , we have that there are at most $\frac{N}{L}$ bad buckets out of N buckets overall (since $S = 1$). Thus, for each bad ball, the probability that it will be thrown into a bad bucket is at most $\frac{1}{L}$. Combining these together, we have that the adversary can win in Game_3 with probability at most

$$\frac{6L^2}{N^3} \cdot \frac{1}{L} = \frac{6L}{N^3} \leq \frac{1}{N^2}$$

where the inequality follows since we assume $X - C \geq 6$ and thus $6L \leq (X - C)L = N$.

Case 2 - $S = 2$: Observing that $\binom{L}{2} = \frac{L(L-1)}{2}$ and recalling that $\binom{X}{t}^{-1} \leq \binom{X}{1}^{-1} = \frac{1}{X} < \frac{L}{N}$, in this case we have

$$\begin{aligned} &\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \\ &< \frac{2}{L(L-1)} \cdot \left(\frac{L}{N}\right)^S \leq \frac{2}{L(L-1)} \cdot \frac{L^2}{N^2} = \frac{2L}{L-1} \cdot \frac{1}{N^2}. \end{aligned}$$

Now, since $S = 2$ we have that at most 2 subarrays were corrupted and so the number of bad buckets from Game_1 is at most $2 \cdot \frac{N}{L}$. Thus, the probability that a bad ball is

thrown into a bad bucket is at most $\frac{2}{L}$, and the probability that the adversary wins in Game_3 is at most $\frac{4}{L-1} \cdot \frac{1}{N^2}$. For $L \geq 5$, we have that $\frac{4}{L-1} \leq 1$ and so the probability that the adversary wins in Game_3 is at most $\frac{1}{N^2}$, as required.

Case 3 – $S = 3$: Observing that $\binom{L}{3} = \frac{L(L-1)(L-2)}{6}$ and using the fact that $\binom{X}{t}^{-1} < \frac{L}{N}$ as above, in this case we have

$$\begin{aligned} & \Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \\ & < \frac{6}{L(L-1)(L-2)} \cdot \left(\frac{L}{N}\right)^3 = \frac{6L^2}{(L-1)(L-2)N} \cdot \frac{1}{N^2} \\ & = \frac{6L}{(L-1)(L-2)(X-C)} \cdot \frac{1}{N^2} < \frac{1}{N^2} \end{aligned}$$

where the last inequality holds since $X \geq L + C$ and so $\frac{L}{X-C} \leq 1$, and since $\frac{6}{(L-1)(L-2)} \leq 1$ when $L \geq 5$ as assumed in the theorem.

Case 4 – $S \geq 4$: Using the bound on Game_1 and again utilizing the fact that $\binom{X}{t}^{-1} < \frac{L}{N}$, we have:

$$\begin{aligned} & \Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \\ & = \binom{L}{S}^{-1} \binom{X}{t}^{-S} \leq \left(\frac{L}{N}\right)^S \leq \frac{L^4}{N^4} = \frac{L^4}{N^2 \cdot (X-C)^2 L^2} \end{aligned}$$

where the last equality is by definition that $N = (X-C)L$. By the assumption that $X > L + C$ we have that $\frac{1}{X-C} < \frac{1}{L}$. Thus,

$$\Pr[\text{Game}_1(\mathcal{A}, X, L, B, C) = 1] \leq \frac{L^4}{N^2 L^4} = \frac{1}{N^2}$$

which suffices since \mathcal{A} must win in Game_1 in order to win Game_3 . ■

An attack for $C = 2$. We proved Theorem 4.4.7 for the case that $C \geq 3$. We conclude this section by showing that when $C = 2$ the theorem does *not* hold and the adversary can win with probability $\frac{2}{N^2}$ (for $B = 2$). The adversary works by corrupting no balls in the second array generated by Game_1 and by corrupting an entire subarray in the first array generated by Game_1 . Specifically, in that execution of Game_1 , it generates $X - 2$ bad balls in some subarray in both arrays that it prepares. Since $C = 2$, the probability that the adversary wins in Game_1 is equals approximately $\frac{2L}{N^2}$. Then, in the later steps of Game_3 the adversary make the first

ball b_1 bad and all the other balls good. Thus, the adversary wins if and only if b_1 is thrown into a bad bucket, which happens with probability $\frac{1}{L}$ (as there are $\frac{N}{L}$ bad buckets). Overall, the winning probability of the adversary is $\frac{2}{N^2}$.

4.4.4 Hash Function Optimization

In [56], the method for validating a multiplication triple using another triple requires the parties to compare their views and verify that they are equal. In this basic comparison, each party sends 3 bits to another party. Since B such comparisons are carried out for every AND gate, this would significantly increase the communication. Concretely, with our parameters of $N = 2^{20}$ and $B = 2$ and our optimizations, this would increase the communication from 7 bits per AND gate to 13 bits per AND gate. In order to save this expense, [56] propose for each party to simply locally hash its view (using a collision-resistant hash function) and then to send the result of the hash only at the end of the protocol. Amortized over the entire computation, this would reduce this communication to almost zero. When profiling Protocol 4 with all of our optimizations, we were astounded to find that these hashes took up almost a *third* of the time in the triples-generation phase, and about 20% of the time in the circuit computation phase. Since the rate of computation is so fast, the SHA256 computations actually became a bottleneck; see Figure 4.3.

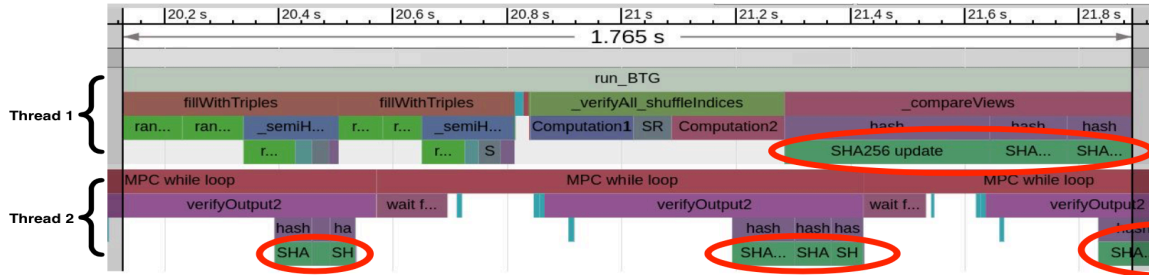


Figure 4.3: Microbenchmarking of Protocol 4.4, using the CxxProf C++ profiler

We solved this problem by observing that the view comparison procedure in [56] requires for each pair of parties to compare their view. The security is derived from the fact that if the adversary cheats then the views of the two *honest* parties are different. As such, instead of using a collision-resistant hash function, we can have each party compute a MAC of their view. In more detail, each pair of parties jointly choose a secret key for a MAC. Then, as the computation proceeds, each party computes a

MAC on its view twice, once with each key for each other party. Then, at the end, each party sends the appropriate MAC to each other party. Observe that the honest parties compute a MAC using a secret key not known to the corrupted party. Thus, the adversary cannot cause the MACs of the two honest parties to have the same tag if their views are different (or this could be used to break the MAC). Note that with this method, each party computes the MAC on its view *twice*, in contrast to when using SHA256 where a single computation is sufficient. Nevertheless, we implemented this using GMAC (optimized using the PCLMULQDQ instruction) and the time spent on this computation was reduced to below 10%. As we show in Section 4.6, this method increases the throughput of the fastest protocol version by approximately 20%.

 Table 4.2: Implementation results; B denotes the bucket size; security level 2^{-40}

Protocol Variant	AND gates/sec	%CPU utilization	Gbps utilization	Latency (ms)
Baseline [56]; Section 4.3 ($B = 3$, SHA)	503,766,615	71.7%	4.55	680
Cache-efficient; Sec. 4.4.1 ($B = 3$, SHA)	765,448,459	64.84%	7.28	623
On-demand; Sec. 4.4.3 ($B = 2$, SHA)	988,216,830	65.8%	6.84	812
On-demand; Sec. 4.4.4 ($B = 2$, GMAC)	1,152,751,967	71.28%	7.89	726
<i>Online-only</i> : on-demand; Sec. 4.4.4 ($B = 2$, GMAC)	1,726,737,312	45.1%	5.11	456.4
<i>Online-only</i> : cache-efficient; Sec. 4.4.1 ($B = 3$, GMAC)	2,132,197,567	41.6%	6.93	367.5

4.5 Trade-off between Security and Efficiency: The Combinatorics of Cut-and-Choose

In the previous sections, we have seen that tight combinatorial analyses are crucial for practical performance. As pointed out in [56], the combinatorial analysis from [25] mandates a bucket-size of $B = 4$ for 2^{20} triples and security level $s = 2^{-40}$. In [56], a tighter combinatorial analysis enabled them to obtain the *same level of security* while reducing the bucket-size from $B = 4$ to $B = 3$. Utilizing a different method, we were further able to reduce the bucket size to $B = 2$. (Combinatorics also played an important role in achieving a cache-efficient shuffle and an on-demand version of the

protocol.) With this understanding of the importance of combinatorics to cut-and-choose, in this section we ask some combinatorial questions that are of independent interest for cut-and-choose protocols.

4.5.1 The Potential of Different-Sized Buckets

We begin by studying whether the use of different-sized buckets can help to increase security. Since our Game_1 (from Section 4.4.1) is specifically designed for the case where all buckets are of the same size, we go back to the more general game of [56] and [25] and redefine it so that buckets may have different sizes. Intuitively, since the adversary does not in advance how many bad balls to choose so that there will be only fully bad buckets, using buckets of different sizes makes it more difficult for him to succeed in cheating. If this is indeed the case, then the winning probability of the adversary can be further decreased, and it may be possible to generate less triples to start with, further improving efficiency. In [56, Theorem 5.3] it was shown that the optimal strategy for the adversary is to make the number of bad balls equal to the size of a single bucket. In this section, we show that even when the buckets sizes are different, the best strategy for the adversary is to make the number of bad balls equal to the size of the smallest bucket. We then use this fact to show that given any set of bucket sizes, changing the sizes so that the bucket sizes of any two buckets differ by at most 1, does not improve the probability that the adversary wins. This makes sense since the adversary's best strategy is to make the number of bad balls equal the size of the smallest bucket, and its hope is that the bad balls will fall into such a bucket. By reducing the gap between buckets (by moving balls from larger buckets to smaller ones) we actually reduce the number of buckets of the smallest size, thereby reducing the probability that all bad balls will be in a bucket of minimal size.

We define a combinatorial game with buckets of different sizes as follows. Let $\vec{B} = \{B_1, \dots, B_N\}$ denote the multiset of bucket sizes where B_i is the size of the i th bucket. As C balls are opened before dividing the balls into buckets, it follows that the overall number of balls generated is $M = \sum_{i=1}^N B_i + C$.

$\text{Game}_4(\mathcal{A}, N, \vec{B}, C)$:

1. The adversary \mathcal{A} prepares M balls. Each ball can be either **bad** or **good**.
2. C random balls are chosen and opened. If one of the C balls is **bad** then output 0. Otherwise, the game proceeds to the next step.

3. The remaining $\sum_{i=1}^N B_i$ balls are randomly thrown into N buckets of sizes $\vec{B} = \{B_1, \dots, B_N\}$.
4. The output of the game is 1 if and only if there exists a bucket B_i that is **fully bad**, and all other buckets are either **fully bad** or **fully good**.

For our analysis we need some more notation. Let B_{\min} be the minimal bucket size. We use $[N]$ to denote the set $\{1, \dots, N\}$. Let $S \subseteq [N]$ be a subset of bucket indices, and let $t_S = \sum_{i \in S} B_i$ be the total number of balls in the buckets indexed by S . Finally, let $n(t) = |\{S \subseteq [N] \mid t_S = t\}|$ be the number of different subsets of buckets such that the number of balls in all buckets in the subset equal exactly t .

We start by computing the winning probability of the adversary:

Lemma 4.5.1. *For every adversary \mathcal{A}_t who chooses t bad balls it holds that*

$$\Pr[\text{Game}_4(\mathcal{A}_t, N, \vec{B}, C) = 1] = \frac{n(t)}{\binom{M}{t}}.$$

Intuitively, for \mathcal{A} to win, the bad balls must fill some subset of buckets (since otherwise there will be a bucket with good and bad balls). Since there are $n(t)$ such subsets, and there are $\binom{M}{t}$ ways to choose t balls out of M balls, it follows that the winning probability of the adversary is $\frac{n(t)}{\binom{M}{t}}$ as stated in the lemma.

Theorem 4.5.2. *If $C \geq B_{\min}$ then for every $S \subseteq [N]$, for every adversary \mathcal{A}_{t_S} who chooses t_S bad balls and for every adversary $\mathcal{A}_{B_{\min}}$ who chooses B_{\min} bad balls, it holds that*

$$\begin{aligned} & \Pr[\text{Game}_4(\mathcal{A}_{t_S}, N, \vec{B}, C) = 1] \\ & \leq \Pr[\text{Game}_4(\mathcal{A}_{B_{\min}}, N, \vec{B}, C) = 1]. \end{aligned}$$

The intuition behind this, is that in order for a subset of buckets to be filled with t bad balls, the smallest bucket in this subset must be filled with bad balls. Thus, it is better for the adversary to choose bad balls for this bucket only, instead for the entire subset.

Next, we show that if \vec{B} that was chosen for the game contains two buckets i and j such that $B_i - B_j > 1$, then moving one ball from the bigger bucket B_i to the

smaller bucket B_j , will result in a game that is more difficult for the adversary to win. This proves that having buckets of significantly different sizes does not improve security, as one can keep moving balls between buckets until all buckets are of size B and $B + 1$ for some B .

Theorem 4.5.3. *Let \vec{B} be a multiset of N bucket sizes that was chosen for the game and assume that there exist $i, j \in [N]$ such that $B_i - B_j > 1$. Let \vec{B}' be a multiset of N bucket sizes obtained by setting*

$$B'_k = \begin{cases} B_i - 1 & \text{if } k = i \\ B_j + 1 & \text{if } k = j \\ B_k & \text{otherwise} \end{cases}$$

If $C \geq B_{\min}$, then for every adversary \mathcal{A}' in the game where \vec{B}' is used, there exists an adversary \mathcal{A} in the game where \vec{B} is used such that

$$\Pr[\text{Game}_4(\mathcal{A}', N, \vec{B}', C) = 1] \leq \Pr[\text{Game}_4(\mathcal{A}, N, \vec{B}, C) = 1]$$

As explained earlier, the intuition behind this is that reducing the gap between large and small buckets in this way can only result in having fewer buckets of smallest size, and therefore the probability that the bad balls will be thrown into a bucket of smallest size can only be reduced.

We conclude that taking different-sized buckets does not improve security (except possibly for the case when exactly two sizes B and $B + 1$ are used). We will use this conclusion in the next section.

4.5.2 Moderately Lowering the Cheating Probability

The discrete cut-and-choose problem. Typically, when setting the parameters of a protocol that has statistical error (like in cut and choose), there is a targeted “allowed” cheating probability which determines a range of values that guarantee the security bound. The parameters are then chosen to achieve the best efficiency possibly within the given range. For example, in a cut-and-choose setting modeled with balls and buckets, the size of the buckets B may be incremented until the security bound is met. However, this strategy can actually be very wasteful. In order to understand why, assume that the required security bound is 2^{-40} and assume that for the required

number of buckets, the bound obtained when setting $B = 3$ is 2^{-39} . Since this is above the allowed bound, it is necessary to increase the bucket size to $B = 4$. This has the effect of increasing the protocol complexity significantly while reducing the security bound to way below what is required. To be concrete, we have proven that the error bound for the protocol version in Section 4.4.1 is $1/N^{B-1}$ (see Theorem 4.4.3). If we require a bound of 2^{-40} and wish to carry out $N = 2^{19} \approx 500,000$ executions, then with $B = 3$ we achieve a cheating probability of only 2^{-38} . By increasing the bucket size to $B = 4$ we obtain a bound of 2^{-57} which is overkill with respect to the desired bound. It would therefore be desirable to have a method that enables us to trade-off the protocol complexity and cheating probability in a more fine-grained manner.

A solution. In this section, we propose a partial solution to this problem; our solution is only partial since it is not as fine-grained as we would like. Nevertheless, we view this as a first step to achieving better solutions to the problem. The solution that we propose in this section is to increment the size of only *some* of the buckets by 1 (instead of all of them), resulting in a game where there are buckets of two sizes, B and $B + 1$. We use the analysis of the previous section to show that this gradually reduces to the error probability, as desired.

Formally, let $\vec{B}^k = \{B_1^k, \dots, B_N^k\}$ be a multiset of bucket sizes such that $B_i^k = B$ for $i \leq k$ and $B_i^k = B + 1$ for $i > k$. In the next lemma, we show that the probability that the adversary wins in the combinatorial game when choosing the bucket sizes in this way is a multiplicative factor of $p = \frac{k}{N}$ lower than when all buckets are of size B . Thus, in order to reduce the probability by $1/2$, it suffices to take $k = N/2$ and increase half the buckets to size $B + 1$ instead of all of them. In the concrete example above, with $N = 2^{19}$ it is possible to reduce the bound to 2^{-40} by increasing half of the buckets to size $B = 4$ instead of all of them, achieving a saving of 2^{18} balls. This therefore achieves the desired goal. We now prove the lemma.

Lemma 4.5.4. *Let $k, N \in \mathbb{N}$ such that $k < N$ and let $p = \frac{k}{N}$. For every bucket-size B , let \vec{B}^k be the multiset of bucket sizes defined as above. Then, for every adversary \mathcal{A}^k in Game_4 where \vec{B}^k is used, there exists an adversary \mathcal{A} in Game_4 where all buckets are of size B such that*

$$\Pr[\text{Game}_4(\mathcal{A}^k, N, \vec{B}^k, C) = 1] \leq p \cdot \Pr[\text{Game}_4(\mathcal{A}, N, B, C) = 1].$$

Proof. In the version of Game_4 with bucket sizes \vec{B}^k , the minimal bucket size is B . Thus using Theorem 4.5.2, it follows that an adversary who chooses B bad balls will maximize its winning probability in both games. Thus, it is sufficient to prove that

$$\begin{aligned} \Pr[\text{Game}_4(\mathcal{A}_B^k, N, \vec{B}^k, C) = 1] \\ \leq p \cdot \Pr[\text{Game}_4(\mathcal{A}_B, N, B, C) = 1] \end{aligned} \quad (9)$$

where \mathcal{A}_B and \mathcal{A}_B^k are adversaries who choose B bad balls in their games. This is sufficient since if Eq. (9) holds then for every \mathcal{A}^k , we can take the adversary \mathcal{A}_B as the adversary for which the lemma holds.

Since there are exactly k buckets of size B , we have that $n(B) = k$ in this game. Furthermore, the number of balls overall is exactly $Bk + (B+1)(N-k) + C$. Thus, by Lemma 4.5.1, it holds that

$$\begin{aligned} \Pr[\text{Game}_4(\mathcal{A}_B^k, N, \vec{B}^k, C) = 1] &= \frac{k}{\binom{Bk + (B+1)(N-k) + C}{B}} \\ &= \frac{p \cdot N}{\binom{BN + (N-k) + C}{B}}. \end{aligned}$$

Similarly, From Lemma 4.5.1, it follows that

$$\Pr[\text{Game}_4(\mathcal{A}_B, N, B, C) = 1] = \frac{N}{\binom{BN + C}{B}}$$

since there are N buckets of size B in this game. Thus, Eq. (9) follows if

$$\frac{p \cdot N}{\binom{BN + (N-k) + C}{B}} \leq \frac{p \cdot N}{\binom{BN + C}{B}}$$

and this holds since $\binom{BN + C}{B} \leq \binom{BN + (N-k) + C}{B}$. \blacksquare

Improving the bound. Observe that the adversary’s winning probability decreases multiplicatively by k/N when $N - k$ balls are added. Thus, in order to reduce the probability by $1/2$ we must add $N - N/2 = N/2$ balls, and in order to reduce the probability by $1/4$ we must add $3N/4$ balls. In general, in order to reduce the probability by $2^{-\zeta}$ we must add $N - N/2^\zeta$ balls. An important question that is open is whether or not it is possible to reduce the probability while adding fewer balls.

4.6 Experimental Evaluation

We implemented the baseline protocol of [56] and the different protocol improvements and optimizations that we present in this paper. (We did not implement the variant in Section 4.4.2 since it has the same efficiency as the variant in Section 4.4.3, and the latter is preferable for practical usage.) All of our implementations use parameters guaranteeing a cheating probability of at most 2^{-40} , as mandated by the appropriate theorem proven above. We begin by describing some key elements of our implementation, and then we present the experimental results.

4.6.1 Implementation Aspects

Parallelization and vectorization. As with [6], our protocol is particularly suited to vectorization. We therefore work in units of 256 bits, meaning that instead of using a single bit as the unit of operation, we perform operations on units of 256 bits simultaneously. For example, we are able to perform XOR operations on 256 bits at a time by writing a “for loop” of eight 32 bit integers. This loop is then automatically optimized by the Intel ICC compiler to use AVX2 256bit instructions (this is called *auto-vectorization*). We verified the optimization using the compiler `vec-report` flag and used `#pragma ivdep` in order to aid the compiler in understanding dependencies in the code. We remark that all of our combinatorial analyses considered “good” and “bad” balls and buckets. All of this analysis remains exactly the same when considering vectors of 256-triples as a single ball. This is because if any of the triples in a vector is bad, then this is detected and this is considered a “bad ball”.

Memory management. We use a common data structure to manage large amounts of triplets in memory efficiently. This structure holds $2^{20} \times 256$ triplets. For triplets $([a], [b], [c])$ (or $([x], [y], [z])$ respectively) we store an array of $2^{20} \times 256$ bits for $[a]$, $2^{20} \times 256$ bits for $[b]$, and $2^{20} \times 256$ bits for $[c]$. This method is known as a *Struct*

of Arrays (SoA) as opposed to an Array of Structs (AoS) and is commonly used in SIMD implementations. It provides for very efficient intrinsic (vectorized) operations, as well as fast communication since we send subarrays of these bit arrays over the communication channel in large chunks with zero memory copying. This reduces CPU cycles in the TCP/IP stack and is open for further optimization using RDMA techniques.

Index shuffling. When carrying out the shuffling, we shuffle indices of an indirection array instead of shuffling the actual triples (which are three 256-bit values and so 96 bytes). Later access to the 256-bit units is carried out by first resolving the location of the unit in $O(1)$ access to the indirection array. This shows substantial improvement as this avoids expensive memory copies. Note that since the triples themselves are not shuffled, when reading the shuffled array during verification the memory access is not serial and we do not utilize memory prefetch and L3 cache. Nevertheless, our experiments show that this is far better overall than copying the three 256-bit memory chunks (96 bytes) when we shuffle data. In Figure 4.5, you can see that the entire cost of shuffling *and verifying* the triples (`_verifyAll_shuffleIndices`) is reduced to less than 30% of the time, in contrast to the original protocol in which it was approximately 55% (see Figure 4.1).

Cache-Aware code design. A typical Intel Architecture server includes a per-core L1 cache (32KB), a per-core L2 cache (typically 512KB to 2MB), and a CPU-wide L3 Cache (typically 25-55MB on a 20-36 core server). L1 cache access is extremely fast at ~ 0.5 ns, L2 access is ~ 7 ns and DDR memory reference is ~ 100 ns. All caches support write back (so updates to cached data is also extremely fast).

We designed our implementation to utilize L1 cache extensively when carrying out the Fisher-Yates shuffling on subarrays. We use two levels of indirection for the index shuffling: the top level of 512 indices and the low level of 2048 indices (under each of the top level indices, yielding 512 subarrays of length 2048 each). As vectors are 1024 byte and 4096 bytes respectively (`uint16` values), they require $1/32$ or $1/8$ of the L1 cache space so L1 will be utilized with very high probability (and in worst case will spill into the L2 cache). This makes shuffling extremely fast. Note that attempting to force prefetch of the index vectors into cache (using `_mm_prefetch` instructions) did not improve our performance, as this is hard to tune in real scenarios.

Offline/online. We implemented two versions of the protocols. The first version

focuses on achieving high throughput and carries out the entire computation in parallel. Our best performance is achieved with 12 workers; each worker has two threads: the first thread generates multiplication triples, and the second carries out the circuit computation. The architecture of this version can be seen in Figure 4.4.

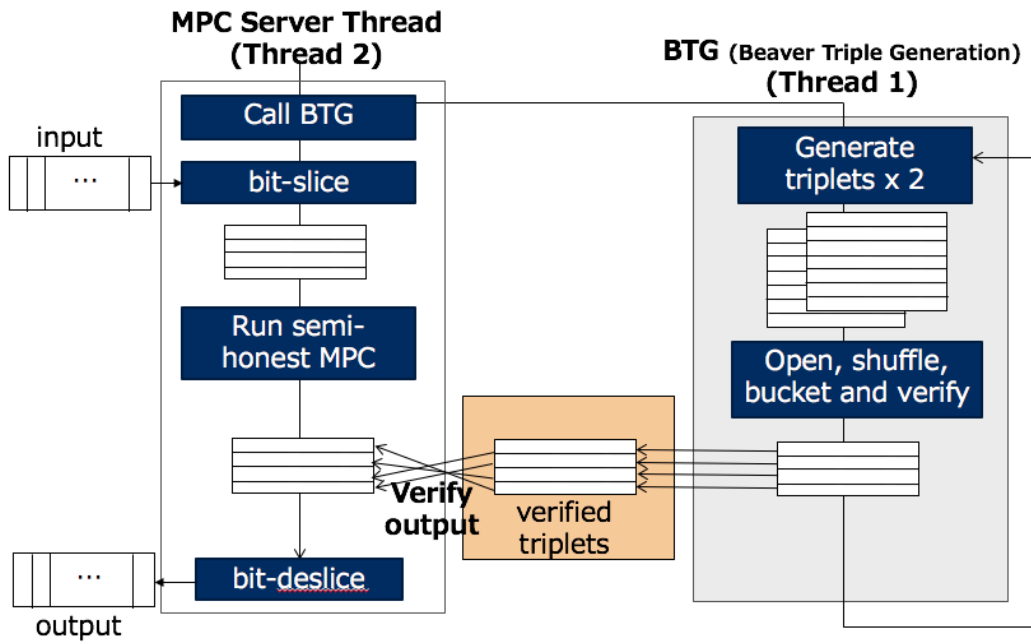


Figure 4.4: Architecture of implementation

The second version focus on achieving fast online performance in an offline/online setting where multiplication triples are prepared ahead of time and then consumed later by a system running only the circuit computation (and verification of that computation). As we have mentioned, the cache-efficient version with bucket-size $B = 3$ is expected to have lower throughput than the version with bucket-size $B = 2$ but lower latency. This is because with $B = 3$ there is no need to randomly choose the triple being used to validate the gate being computed. We therefore compared these; note that in both cases we used the GMAC optimization described in Section 4.4.4 so that we would be comparing “best” versions.

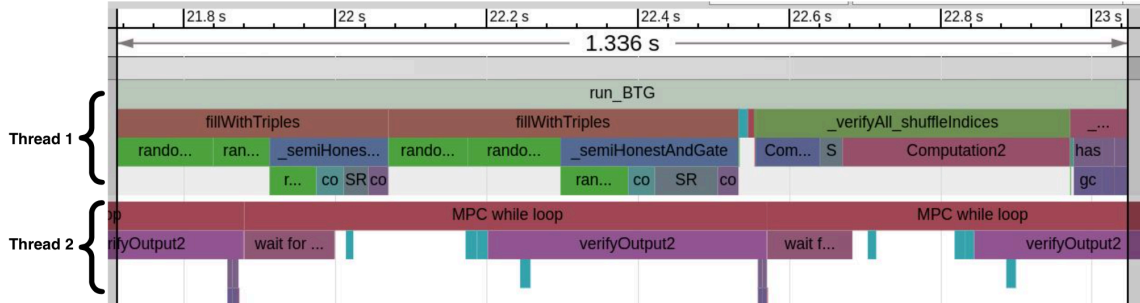


Figure 4.5: Microbenchmarking of best protocol variant, using the CxxProf C++ profiler (run on a local host)

4.6.2 Results and Discussion

We ran our implementations on a cluster of three mid-level servers connected by a 10Gbps LAN. Each server has two Intel Xeon E5-2650 v3 2.3GHz CPUs with a total of 20 cores. The results appear in Table 4.2. Observe that each of the protocol improvements presented here provides a dramatic improvement:

- **Section 4.4.1:** Replacing the naive Fisher-Yates shuffle on an array of size 2^{20} with our cache-efficient shuffle yields an increase of about 50% in throughput;
- **Section 4.4.3:** Reducing the communication (in addition to the cache-efficient shuffle) by reducing the bucket-size from $B = 3$ to $B = 2$ and randomly choosing triples to verify the circuit multiplications yields a further increase of about 30%. (This is as expected since the reduction in communication is exactly 30%.)
- **Section 4.4.4:** Replacing the use of SHA256 with the GMAC computations yielded an additional increase of over 15%.

Our best protocol version has a throughput of about **2.3 times** that of baseline version. This result unequivocally demonstrates that it is possible today to achieve *secure computation with malicious adversaries at rates of well over 1-billion gates per second* (using mid-level servers).

It is highly informative to also consider the results of the online-only experiments (where triples are prepared previously in an offline phase). As expected, the protocol version with bucket-size $B = 3$ is better in the online phase since no random choice of triples is needed. The throughput of the best version exceeds *2 billion AND gates*

per second. Importantly, latency is also significantly reduced to 367.5ms; this can be important in some applications.

Microbenchmarks. Microbenchmarking of the faster protocol can be seen in Figure 4.5. In order to understand this, see Figure 4.4 for a description of the different elements in the implementation. The `run_BTG` thread generates multiplication (Beaver) triples. Each triple is generated by first generating two random sharings and then running a semi-honest multiplication. After two arrays of triples are prepared (since we use buckets of size $B = 2$), they are verified using the `_verifyAll_shuffleindices` procedure; this procedure carries out shuffling and verification. The second thread runs MPC computation to compute the circuit, followed by verifying all of the multiplications in the `verifyOutput2` procedure.

Part II

Applications: How to Realize MPCs for Complex Functionalities — Bridging from Efficient Primitives to Efficient Applications

Chapter 5 Application (1): Generalized SPDZ Compiler for MPC based on Secret Sharing

5.1 Introduction

In recent years there has been immense progress in the efficiency of MPC protocols, and today we can securely compute large Boolean and arithmetic circuits representing real computations of interest. However, most MPC protocols rely on *circuit-based* approach. Namely, these protocols require the description of a Boolean and/or arithmetic circuit in order to run. This is a significant obstacle in the deployment of MPC, since circuits for real problems of interest can be very large and very hard to construct. No matter how much the efficiency of MPC improves, it will be difficult to spread MPC socially unless this issue is solved. In order to deal with this issue, there has been quite a lot of work on compiling high-level programs to circuits.

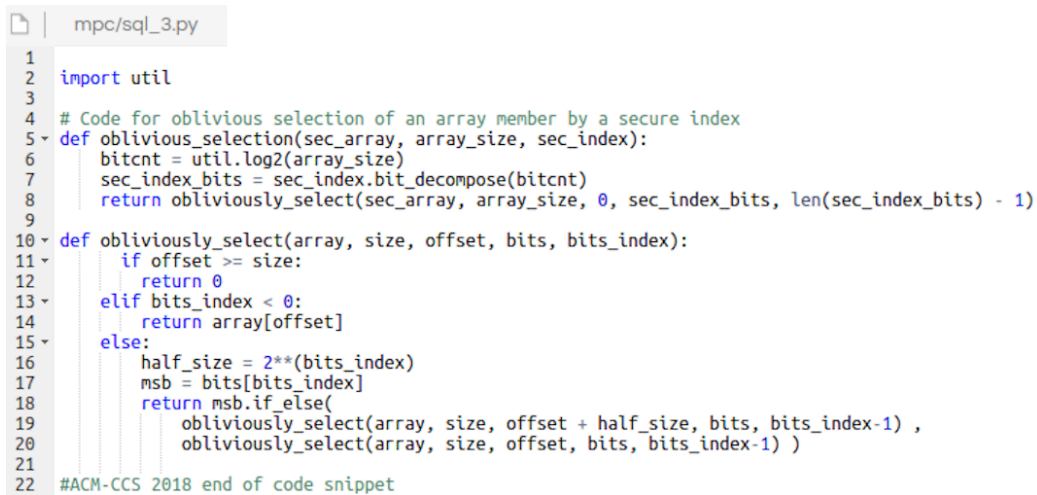
5.2 Related Work

There are a lot of work on the MPC compilers [53, 117, 26, 27]. Unfortunately, many of these works are limited in the size of the circuit that they can generate, and most of them do not deal with the general problem of combined arithmetic and non-arithmetic (Boolean) computations. In addition, the paradigm of working with static circuits is problematic for huge computations, due to the size of the circuit that must be dealt with (this issue has been considered in [117] and elsewhere, but can still be an issue).

In contrast to the above, the series of works called “SPDZ” took a very different approach. SPDZ is the name of a specific protocol for honest-minority multi-party computation [43]. However, beyond improvements to the protocol itself, follow-up work on SPDZ included the implementation of an extremely powerful MPC run-time environment/compiler that is integrated into the SPDZ low-level protocol [42, 72, 24]. From here on we differentiate between the *SPDZ protocol* which is a way of executing

secure MPC over arithmetic circuits, and the *SPDZ compiler* that is a general run-time environment that takes code written in a high-level Python-type language, and executes it in MPC over the SPDZ protocol. We stress that SPDZ does not generate a circuit and hand it down to the low-level protocol. Rather, it behaves more like an interpreter, dynamically calling the lower-level protocol to carry out low-level operations.

The SPDZ Protocol and Compiler A key property of the SPDZ compiler is that it separates the basic operations provided by MPC protocols (binary or arithmetic circuits) from a protocol (or program) using those operations as building blocks. While the basic operations mostly consist of simple arithmetic over some ring (more precisely, a field in case of SPDZ), combining them to achieve higher-level operations, like integer or fixed-point division, is a more complex matter. However, integrating such higher-level operations into the core MPC engine is not a good strategy because the reduction to basic operations is likely very similar even for different underlying protocols. The SPDZ compiler provides a tool to write more complex building blocks, which then can be used in arbitrary MPC applications without being concerned about the details of those blocks nor the underlying protocol. A concrete example of the ease in which complex secure computations can be specified appears in Figure 5.1.



```

1  import util
2
3  # Code for oblivious selection of an array member by a secure index
4  def oblivious_selection(sec_array, array_size, sec_index):
5      bitcnt = util.log2(array_size)
6      sec_index_bits = sec_index.bit_decompose(bitcnt)
7      return obliviously_select(sec_array, array_size, 0, sec_index_bits, len(sec_index_bits) - 1)
8
9
10 def obliviously_select(array, size, offset, bits, bits_index):
11     if offset >= size:
12         return 0
13     elif bits_index < 0:
14         return array[offset]
15     else:
16         half_size = 2**(bits_index)
17         msb = bits[bits_index]
18         return msb.if_else(
19             obliviously_select(array, size, offset + half_size, bits, bits_index-1) ,
20             obliviously_select(array, size, offset, bits, bits_index-1) )
21
22 #ACM-CCS 2018 end of code snippet

```

Figure 5.1: SPDZ Python code for oblivious selection from an array.

This program describes the task of selecting an element from an array, where both

the array values *and* the array index are private (and thus shared). Given that the size of the array is also a variable, this is very difficult to specify in a circuit. This highlights another huge advantage of this paradigm. The SPDZ system facilitates modular programming techniques, enabling the software engineer to program functions that can be reused in many programs. (Note that a simpler linear program could be written for the same task, but this method is more efficient. Observe the richness of the language, enabling recursion, if-then-else branching, and so on.)

Extending the SPDZ compiler. Prior to our work, the SPDZ compiler was closely integrated with the SPDZ low-level protocol, preventing its more broad use. The primary aim of this work is to extend the SPDZ compiler so that other protocols can be integrated into the system with ease. This involved making changes to the SPDZ compiler at different levels, as is described in Section 5.4. In order to demonstrate the strength of this paradigm, we integrated three different protocols of completely different types. Specifically, we integrated the honest-majority multi-party protocol of [85] for arithmetic circuits over a field, the three-party honest-majority protocol of [6, 5] for arithmetic circuits over the ring \mathbb{Z}_{2^n} for any n , and the BMR protocol [13, 88] for constant-round multi-party computation for Boolean circuits. The integration of the former protocol required the fewest number of changes, since it works over any field just like the original SPDZ, whereas the other protocols required more changes. For example, the SPDZ compiler already comes with high-level algorithms for fixed-point and floating point operations, integer division and more. All of these are reusable as-is for any other protocol based on fields. However, for protocols over the ring \mathbb{Z}_{2^n} , different high-level algorithms needed to be developed. We have done this, and thus other protocols over rings can utilize the relevant high-level algorithms.

We stress that the focus of our extensions were not to integrate these specific protocols, but to modify the SPDZ system in order to facilitate easy integration of other protocols by others. We believe that this is a significant contribution, and will constitute a step forward to enabling the widespread use of MPC.

Bit decomposition and ring composition. The advantage of working over arithmetic circuits (in contrast to Boolean circuits) is striking for computations that require a lot of arithmetic, as is typical for computing statistics. In these cases, addition is for free, and multiplication of large values comes at a cost of a single operation. However, most real-world programs consist of a combination of arithmetic and non-arithmetic

computations, and thus need a mix of arithmetic and Boolean low-level operations. In order to facilitate this, it is necessary to have *bit decomposition* and *ring composition* operations, to convert a shared field/ring element to a series of shares of its bit representation and back. This facilitates all types of computation, by moving between the field/ring representation and bit representation, depending on the computation. For example, consider an SQL query which outputs the average age of homeowners with debt above the national average, separately for each state. This requires computing the national debt average (arithmetic), comparing the debt of each homeowner with the national average (Boolean) and computing the average age of those whose debt is greater (mostly arithmetic for computing the sum, and one division for obtaining the average). Note that the last average requires division since the number of homeowner above the average is not something revealed by the output, and division is computed using the Goldschmidt method which requires a mix of arithmetic and bit operations, including conversions.

As we discuss below in Section 5.3, the SPDZ compiler includes high-level algorithms for many complex operations, and as such includes bit decomposition and ring composition. In some cases, the operations rely on division in the field and so cannot be extended to rings. In order to facilitate working with rings, we therefore develop novel protocols for bit decomposition and ring composition between \mathbb{Z}_{2^n} and \mathbb{Z}_2 that are based on replicated secret sharing and therefore compatible with [6, 5]. Since \mathbb{Z}_{2^n} preserves the structure of the individual bits much more than \mathbb{Z}_p for a prime p , it is possible to achieve *much faster* decomposition and composition than in the field case. Thus, in programs that require a lot of conversions, ring-based protocols can way outperform field-based protocols. However, field-based protocols are typically more efficient for the basic arithmetic (e.g., compare the ring version of [5] to [85]). Thus, different low-level protocols have different performance for different programs. Stated differently, there is no “best” protocol, even considering a specific number of parties and security level, since it also depends on the actual operations carried out (this is also true regarding deep vs shallow circuits, and constant versus non-constant round protocols). This gives further justification to have a unified SPDZ system that can work with many low-level protocols *of different types*, so that a program can be written once and tested over different protocols in order to choose the best one.

Our protocols for bit decomposition and ring composition are described in Section 6.1.

Implementation and experiments. In Section 5.5, we present the results of experiments we ran on programs for evaluating unbalanced decision trees (this is more complex than balanced decision trees due to the need for the evaluation to be completely oblivious) and for evaluating complex SQL queries. Although the focus of our work is not efficiency, we report on running times and comparisons in order to provide support for the fact that this SPDZ extension is indeed very useful and meaningful.

Our implementation is open-source and available for anyone interested in utilizing it.

5.3 Review on the SPDZ Protocol and Compiler

5.3.1 Overview

SPDZ is the name given to a multi-party secure computation protocol by Damgård et al. [43, 42] that works for any n parties. It provides active (malicious) security against any $t \leq n$ corrupted parties, and it works in the preprocessing model, that is, the computation is split into a data-independent (“offline”) and a data-dependent (“online”) phase. The main idea of SPDZ is to use relatively expensive somewhat homomorphic encryption in the offline phase while the online phase purely relies on cheaper modular arithmetic primitives. This also allows for an optimistic approach to the distributed decryption used in the offline phase: Instead of proving correct behavior using zero-knowledge proofs, the parties check the decrypted value for correctness and abort in case of an error. Nevertheless, there is no leakage of secret data because no secret data has yet been used.

The main link between the two phases is a technique due to Beaver [10], which reduces the multiplication of secret values to a linear operation on secret values using a precomputed multiplication of random values and revealing of masked secret-shared values. Using a LSS makes this technique straightforward to use. Additive secret sharing is trivially linear, and it provides the desired security against any number of $t \leq n$ corrupted parties. On the top of additive secret sharing, SPDZ also uses an information-theoretic tag (the product of the secret value and a global secret value), which is additively secret-shared as well, thus preserving the linear property.

Keller et al. [72] have created software to run the online phase of any computation, optimizing the number of communication rounds. The software receives a description of the computation in a high-level Python-like language, which is then compiled into a concise byte-code that is executed by the SPDZ virtual machine (which includes the actual SPDZ MPC protocol); see Figure 5.2. The design of the virtual machine follows the design principles of processors by providing instructions such as arithmetic over secret-shared or public values (and a mix between them), and branching on public values. The inclusion of branching means that one can implement concepts common in programming languages such as loops, if-else statements, and functions. While the conditions for loop and if statements can only depend on public values,¹ this provides an obvious benefit in reducing the representation of a computation and the cost of the optimization described below. In particular, it is possible to loop over a large set of inputs without representing the whole circuit in memory. We call this software layer the **SPDZ compiler**, in order to distinguish it from the **SPDZ protocol**. We remark that although the SPDZ compiler was developed with the SPDZ protocol specifically in mind, its good design enabled us to extend it to other protocols and make it a general MPC tool.

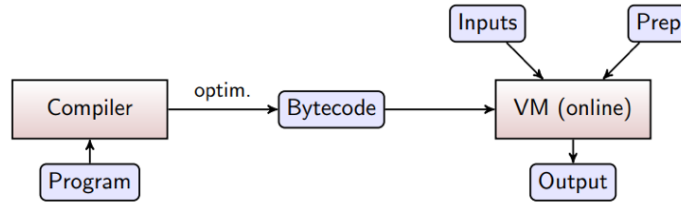


Figure 5.2: High-level SPDZ compiler architecture

5.3.2 Circuit Optimizations

The core optimization of the software makes use of the fact that, using Beaver’s technique, the only operation that involves communication is the revealing of secret values. This means that the compiler can merge all operations in a single communication round into a single opening operation, effectively reducing the communication to the minimum number rounds for a given circuit description. In addition, the software

¹This is an inherent requirement for “plain” multi-party computation. There are solutions that overcome this [68], but they come with a considerable overhead.

splits the communication into two instructions to mark the sending and the receiving of information, and optimizes by placing independent computations in-between the send-and-receive, providing the ability to use the time that is required to wait for information from the other parties. As such, all opening operations are framed between **start** and **stop** instructions, and independent instructions that can be processed in parallel to the communication are placed between them. For example, **startopen** denotes the beginning of a series of instructions to open (reveal) shares, and **stopopen** denotes the end of the series.

In order to achieve the above effect of grouping all communication messages per round, the SPDZ compiler represents the computation as a directed acyclic graph where every instruction is represented as a node, and nodes are connected if one instruction uses another's output as input. The vertices are assigned weight *one* if the source instructions start a communication operation and *zero* otherwise. The communication round of any instruction is then the longest path from any source with respect to the vertex weights. It is straight-forward to compute this by traversing the instructions in order and assigning the maximum value of all input vertices to each instruction.

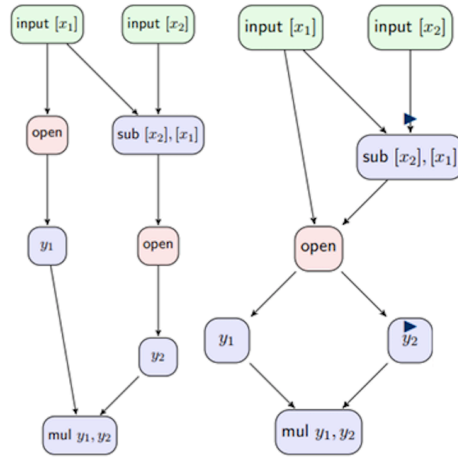


Figure 5.3: Representing a program as a directed acyclic graph

It is important to note that merging all instructions that can be run in parallel needs to be done carefully. In particular, it does not suffice to merge the open instructions that are independent of each other, but also any operations that the open

depends on. This is solved by computing the topological order of the changed graph, and by adding vertices between instructions with side effects, in order to maintain the order between them. This results in a trade-off because adding more vertices in order to preserve the order can lead to more communication rounds.

The optimization described in this subsection (of reducing the number of communication rounds) is only possible on a straight-line computation without any branching. We therefore perform this optimization separately on each part of the computation of maximal size. These components are called basic blocks in the compiler literature.

5.3.3 Higher-Level Algorithms

In terms of arithmetic operations, the virtual machine provides algebraic computations on secret values provided by the MPC protocol (addition and multiplication) and general field arithmetic on public values (such as addition, subtraction, multiplication, and division). This clearly does not suffice for a general, easy to use programming interface. Therefore, in addition to the Beaver’s technique for secret-value multiplication described above, the compiler comes with a library that provides non-algebraic operations on secret values such as *comparison* (equals, less-than, etc.), and arithmetic for both floating- and fixed-point numbers. This library is based on a body of literature [31, 30, 4] that uses techniques such as statistical masking to implement such operations without having to rely solely on field-arithmetic circuits.²

The following bit decomposition of a secret-sharing of $0 \leq x < 2^m$ for some m illustrates the nature of these protocols. Assume that $[x]$ is a secret-sharing of a x in a field \mathbb{F} such that $2^{m+k} < |\mathbb{F}|$, with k being the statistical security parameter. Let r be a random value such that $0 \leq r < 2^{m+k}$, consisting of bits r_i for $i = 0, \dots, m+k-1$, and let $[r_0], \dots, [r_{m+k-1}], [r]$ be their secret sharings, all over the field \mathbb{F} . (It is possible to generate these shares by sampling $[r_0], \dots, [r_{m+k-1}]$ in the offline phase, as discussed in [42], and then computing $[r] = \sum [r_i] \cdot 2^i$ locally.) Similarly, we can compute $[z] = [x + r]$ from $[r]$ and $[x]$ locally. Observe that z statistically hides x because the statistical distance between the distributions of z and of r is negligible in k . Therefore, we can reveal z and decompose it into bits z_0, \dots, z_{m+k-1} . Finally, the shares of the bits of x , $([x_0], \dots, [x_{m-1}])$, can be computed from (z_0, \dots, z_{m+k-1}) and

²Arithmetic circuits are essentially polynomials, and a naive implementation of an operation like the comparison of numbers in a large field is very expensive.

$([r_0], \dots, [r_{m+k-1}])$ via a secure computation of a Boolean circuit.

The compiler also provides the same arithmetic interface when using the SPDZ protocol with a finite field of characteristic two, allowing the execution of the same computation on different underlying protocols. We used this as a stepping stone for the extension using GCs below because of the similarity between them.

Implementing these algorithms at this level rather than within the virtual machine below has the advantage that all optimizations in the compiler are automatically applied to any MPC VM. We stress that these algorithms are part of the SPDZ compiler layer.

5.4 Software Design and Implementation for Making SPDZ a General Compiler

In order to generalize the SPDZ compiler to work for other protocols, modifications needed to be made at multiple levels. Our aim when designing these changes was to make them as general as possible, so that other protocols can also utilize them. We incorporated three very different protocols in order to demonstrate the generality of the result:

1. *Honest-majority MPC over fields*: We incorporated the recent protocol of [85] that computes arithmetic circuits over any finite field, assuming an honest majority. This protocol has a direct multiplication operation, and does not work via triples like the SPDZ protocol. (The protocol does use triples in order to prevent cheating, but not in a separate offline manner.) The specific protocol incorporated works over \mathbb{Z}_p with Mersenne primes $p = 2^{61} - 1$ or $p = 2^{127} - 1$.
2. *Honest-majority MPC over rings*: We incorporated the three-party protocol of [6, 5] that computes arithmetic circuits over any *ring* including the ring \mathbb{Z}_n of integers for any $n \geq 1$. The fact that this protocol operates over a ring and not a field means that it is not possible to divide values; this requires changing the way many operations are treated, as will be described below.
3. *Honest minority MPC for Boolean circuits*: We incorporated a protocol for computing any Boolean circuit using the BMR paradigm [13]. Our starting point for this purpose was the software of [74] that was developed for a different purpose;

we therefore made the modifications needed for our purpose.

We discuss these different protocols in more detail below. In this section, we describe the changes that we made to the SPDZ compiler in order to enable other protocols to be incorporated in it, with specific examples from the above.

5.4.1 Modifications to the SPDZ Compiler

Infrastructure modifications at the compiler level. The main difficulty in adapting the SPDZ compiler to other protocols lies in the fact that many protocols do not use the Beaver technique, and so do not reduce secret value multiplications to the opening of masked values only. Such protocols include secret-value multiplications as an atomic operation of the protocol, and thus working via `startopen` and `stopopen` only would significantly reduce the protocol’s performance.³ We therefore generalized the communication pattern of the compiler to allow general communication and arbitrary pairs of start/stop instructions for communication, rather than specifically supporting only start/stop of share openings. Our specific protocols have atomic multiplication operations that involve communication, and so we specifically added `e_startmult` and `e_stopmult` (which are start and stop of multiplication operations, where the `e`-prefix denotes an extension), but our generalization allows adding any other type of communication as well.

Since the multiplication within the protocols that we added involves communication, it is desirable to merge as many multiplications as possible with the reveal (or open) operations in the SPDZ compiler. The fork of the SPDZ compiler used by Keller and Yanay for their BMR implementation [73, 74] provides functionality to merge several kinds of instructions separately (AND and XOR in their case). We used this for multiplication and open instructions, resulting in circuit descriptions that minimize the number of multiplication and open rounds separately. This is not optimal because it does not provide full parallelization of the communication incurred by multiplication and open operations that could be carried out in parallel. However, we argue that this is sufficient because in protocols that support atomic multiplication, opening is typically only necessary at the end of a computation that involves many rounds of multiplications.

³We stress that the only communication in the SPDZ protocol is in the opening of shared values, and all other operations – including multiplication – are reduced to local computation and opening.

Algorithm modifications at the compiler level. The modular construction of the compiler and the algorithm library allows us to re-use many higher-level protocols mentioned in the previous section. These algorithms are represented in the compiler as expansions of an operation. For example, multiplication of shared values in the SPDZ protocol is a procedure that utilizes a multiplication triple, and carries out a series of additions, subtractions and openings in order to obtain the shared product. This algorithm is replaced by a single `startmult` and `stopmult` using our new instructions for low-level MPC protocols that have an atomic multiplication operation. See Figure 5.4 for code comparison.

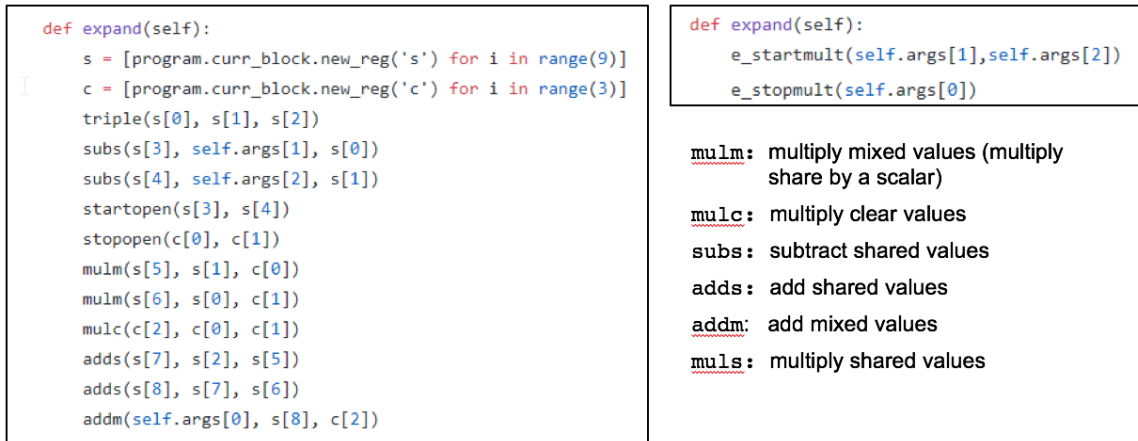


Figure 5.4: Multiplication in the original SPDZ compiler vs using new instruction extension

For the case of our honest-majority field-based protocol, this is the only change that we needed to make to the compiler. This is because all of the original SPDZ compiler algorithms (e.g., for floating and fixed-point operations, integer division, bit decomposition, etc.) work for any field-based MPC, and thus also here. However, when field division is not available, as in the example of the ring-based protocol, different high-level algorithms needed to be provided. A very important example of this relates to bit decomposition and ring composition for ring-based protocols, which is an operation needed for many higher-level arithmetic operations including non-algebraic operations like comparison. We present a new highly-efficient method for bit decomposition and ring composition over \mathbb{Z}_{2^n} in Section 6.1, and this was

incorporated on the algorithm level. In addition, new algorithms were added for fixed-point multiplication and division, integer division, comparison, equals, and more. We stress that once the infrastructure modifications were made, all of these changes are algorithmic only, meaning that they rewrite the **expand** operation that converts a high-level algorithm into a series of low-level supported operations (like multiplication in Figure 5.4).

Modifications to bytecode. The bytecode that is generated by the compiler includes the low-level instructions and opcodes supported by the MPC protocol itself. As such, some changes were needed to add new instructions and opcodes supported by the other MPC protocols. Thus, a direct multiplication opcode needed to be added (for both the field and ring protocols), as well as some additional commands for the bit decomposition and ring composition needed for the ring protocol (e.g., the local decomposition steps described in Sections 6.1.3 and 6.1.4). Finally, a new **verify** command was added since the honest-majority field and ring protocols do not use SPDZ MACs and verify correctness in a different way. The bytecode also includes a lot of instructions needed for jumping, branching, merging threads and so on. Fortunately, all of this can be reused as is, without any changes.

Modifications to the virtual machine. On the level of the virtual machine, we have modified the SPDZ compiler software to call the relevant function of an external library for every instruction that involves secret-shared values. This comes down to roughly twenty instructions. We have done this in a way that facilitates plugging in other backend libraries, which allows us to easily run the same program using different protocols. This is in line with our goal of enabling the same high-level interface to be used to program for completely different MPC schemes. For the field case, the changes here were relatively small. The multiplication was changed, but so was scalar addition since in the SPDZ protocol each party carries out the same operation locally, whereas different parties act differently for scalar addition in the replicated secret-sharing protocol version of [85]. In addition, triple generation is not carried out offline but done on demand, and the MAC was disabled at the VM level (i.e., an extension was added to optionally disable the MAC so that the VM is compatible both with protocols that use and do not use MACs). Finally, the original SPDZ protocol relies on Montgomery multiplication [94]. While this is efficient for general moduli, in some cases like when using Mersenne primes, more efficient modular multiplication can be

achieved directly. The field protocol implementation of [85] utilizes Mersenne primes, and this was therefore also integrated into the VM interface.

For the ring protocol, there were more changes required since the division of clear elements is not supported in a ring, and since more instructions are needed at the basic protocol level (for decomposition and ring composition, as described above). In addition, the `input` procedure was changed since the secret sharing is different. We stress that these are not just at the protocol level since the VM uses special registers for local operations (to improve performance) and so these need to be modified.

We remark that the compiler, bytecode, and VM needed to be very significantly modified for the Boolean circuit (BMR) protocol, and thus a separate branch was created. This is understandable since the protocol is of a completely different nature. Nevertheless, the key property that it *all runs under the same MPC program high-level language* is achieved, and thus to the “MPC user” writing MPC programs, this is not noticeable.

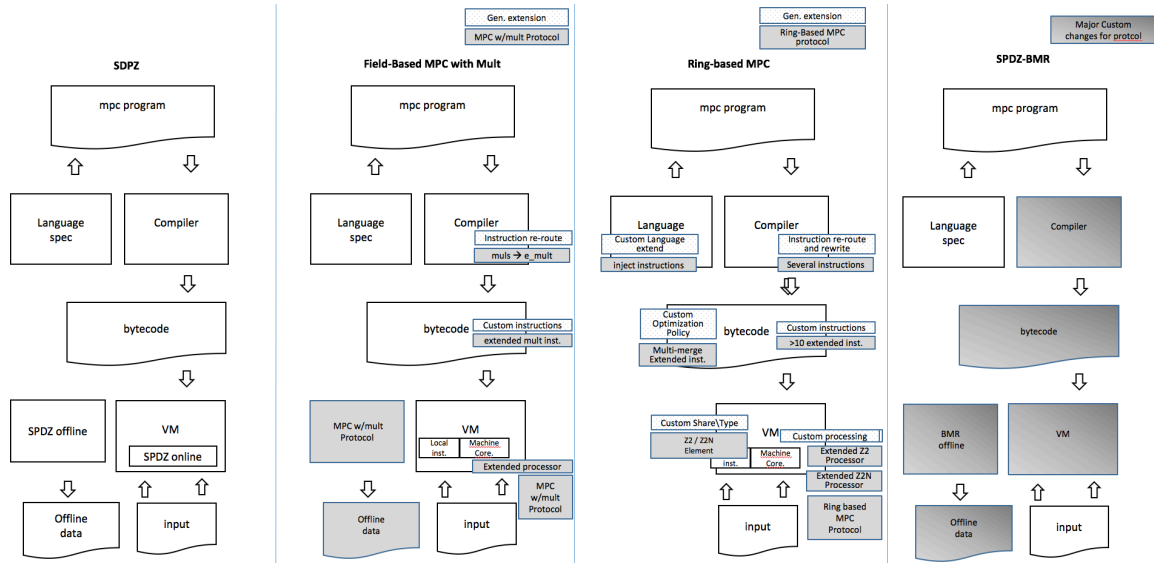


Figure 5.5: The extensions applied to the SPDZ compiler of [42]

Explanation of Figure 5.5. In Figure 5.5 we present a diagram illustrating the different extensions to the SPDZ compiler, for all three protocols incorporated. On the left, the original SPDZ architecture is presented. Then, the field-based protocol of [85] is presented, with relatively minor changes (mainly adding the multiplication extension); of course, the MPC protocol at the lower level is replaced as well. Next,

the ring-based protocol is presented, and it includes more modifications, including support for different types of shares and numbers, as well as more modifications to the compiler and below. There are also some additions to the language itself, since adding explicit instructions like `inject` (which maps a bit into a ring element) improves the quality of the compiler. Finally, the architecture for the BMR protocol is added; as stated above, this requires major changes throughout, except for the programmer interface and language which remain the same.

5.4.2 Incorporating BMR Circuits

In this section, we provide additional details about the incorporation of the BMR Boolean circuit protocol into SPDZ. Since the original SPDZ protocol is based on *secret sharing* and *arithmetic circuits*, the changes required to incorporate a garbled-circuit based protocol were the most significant.

In order to evaluate our programs in a GC setting, we have made use of the recently published software implementing oblivious RAM [73, 74] in the SPDZ-BMR protocol [88]. The latter denotes the combination of BMR, which is a method of generating a GC using any MPC scheme, with the SPDZ protocol [43] as the concrete scheme. While there are recent protocols achieving similar goals [121, 64], we would argue that the BMR software is the most powerful one publicly available to date, and that it still gives a reasonable indication of the performance of GCs with active security.

The software follows the same paradigm as SPDZ in that it implements a virtual machine that executes bytecode consisting of instructions for arithmetic, branching, input/output, etc. The main difference is that arithmetic here means XOR and AND. Furthermore, while the smallest units at the virtual machine level are secret-shared and public values in a field for SPDZ, here they are vectors of secret-shared bit and public values. This leads to more concise circuit descriptions. Furthermore, the compiler merges several types of instructions to further vectorize instructions, which may reduce the number of communications rounds (e.g., for inputs), enable the use of several processor cores, and facilitate pipelining of AES-NI instructions when evaluating as many AND gates in parallel as possible.

The primary goal of the software of [73, 74] is the evaluation of ORAM. Hence we needed to extend it in various aspects, most notably the following:

Private inputs: This feature was omitted from [73, 74] who wished only to evaluate

the performance of computation. In our context however, private inputs play a major role. This change mostly affected the virtual machine of the BMR implementation.

Arithmetic: While the software of [73, 74] contains provisions for integer arithmetic in fields of characteristic two (and thus for binary circuits) and for fixed-point calculations in arithmetic circuits, we had to combine and supplement this for our purposes. In particular, it turned out that the translation of fixed-point division from arithmetic to binary circuits is non-trivial because keeping exact track of bit lengths is vital in the latter. Since the virtual machine only deals with binary circuits by design, this change was exclusively on the compiler side.

The software is incomplete in the sense that it only implements the evaluation phase securely, while the use of the SPDZ protocol in the garbling part is simulated using a separate program. Nevertheless, the evaluation timings are accurate because the GC is read from solid-state disks. Furthermore, the uniform nature of the circuit generation as well as the offline phase of SPDZ (called function-dependent phase in this context) allowed us to micro-benchmark the two phases. For the latter, this has been done in various previous works [71, 42].

5.5 Experimental Evaluation

5.5.1 Implementation Aspects

In order to evaluate our toolchain and protocol, we have implemented various computations, ranging from a simple mean and variance computations, to a more involved computations of inference via a non-balanced decision tree and the private processing of an SQL query. The SQL query is quite a complex computation and is derived from the following query for a typical survey:

```
SELECT count(*), avg(credit limit) FROM Census
WHERE State=Utah
GROUP BY Age, Sex HAVING count(*) > 100;
```

This query computes the average credit limit of every age-group and sex (i.e., average credit limit of 30 year old females, average credit limit of 30 year old males, and so on), outputting only results for sets that have at least 100 data items in the set. This last requirement is necessary to preserve privacy and to ensure that there are

no results based on very few individuals. For all fields that have a small range such as state, age, and sex we input the data in a bit-wise unary encoding (a list of bits of which only one is 1), which simplifies the selection operation in secure computation.

The decision tree private inference example uses the decision tree built from real data published for a paper on credit decisions [116]. The concrete decision tree in our computation has 1256 leaves at depths from 4 to 30. Since multi-party computation reveals the amount of computation (i.e., how many gates are computed), we have to always execute 30 decisions in order to hide the path traversed in the evaluation. This is achieved using dummy data if a leaf is reached before the last step. Furthermore, traversal of the tree makes use of oblivious selection from the current depth of the tree represented as an array (this selection is of the node to be used in the current level of the decision tree), in order to not reveal anything about the path of computation in the tree.

Whenever non-integer computation is required, we use fixed-point computation as implemented in the SPDZ compiler [24]. This is justified because the mean over a set of numbers in a limited range will also be in this range and thus not require the larger range of floating-point numbers. The bit decomposition and ring composition used for the SQL query is the SPDZ compiler method for SPDZ and $\text{MHM}\mathbb{Z}_p$, and is our new method from Section 6.1. The times given are for the basic conversion (see Table 6.8) which minimizes the amount of communication at the expense of a higher number of rounds. (We also implemented the other versions, but they were slower in our tests since we ran the experiments on a very low-latency network.)

We ran our experiments on AWS with three parties in a single region, using `m5.12xlarge` instances providing 10 Gbps network communication. The only exception is for the BMR protocol, where we used `i3.2xlarge` instances due to the increased amount of storage needed to store the GC.

5.5.2 Results and Discussion

Figures 5.6–5.8 show the online times for mean, variance, and our SQL query for various numbers of inputs, and Table 5.1 shows the results of decision tree computation. $\text{MHM}\mathbb{Z}_p$ refers to the malicious honest-majority protocol over \mathbb{Z}_p of [85], while $\text{SHM}\mathbb{Z}_{2^n}/\mathbb{Z}_2$ refers to the semi-honest honest-majority protocol of [6] over the ring of integers \mathbb{Z}_{2^n} for any $n \geq 1$. Note that the different protocols operate in different

security models: SPDZ and BMR provide security in the presence of any $t \leq n$ malicious corruptions, MHM \mathbb{Z}_p provides security in the presence of a malicious minority, and SHM $\mathbb{Z}_{2^n}/\mathbb{Z}_2$ provides security in the presence of semi-honest adversaries with an honest minority. This explains the expensive offline phase for SPDZ and BMR because more expensive operations such as somewhat homomorphic encryption are used there. To time the offline phase of SPDZ, the newer “Low Gear” protocol [71] has been used on **r4.8xlarge** instances due to the larger memory requirement of homomorphic encryption, while the MASCOT protocol [70] has been used for BMR. (The SPDZ offline was also computed using a large number of threads, in contrast to a single thread for MHM/SHM.)

We stress that we present these results to demonstrate the new capability of writing a single complex program and running it on *four completely different low-level protocols*, and not in order to compare efficiency. Indeed, we are continuing our work to improve the efficiency of the SPDZ-compiled lower-level protocols (e.g., adding vectorization, more parallelism and specific optimizations). Nevertheless, it is interesting to observe that the SHM \mathbb{Z}_{2^n} method is approximately *50 times faster* than the MHM \mathbb{Z}_p method for the SQL processing (Figure 5.8). Although there is a difference between semi-honest and malicious, the cost of MHM \mathbb{Z}_{2^n} is only 7-times slower than SHM \mathbb{Z}_{2^n} [5]. The rest of the difference is due to the faster bit decomposition and ring composition for the ring-based protocol versus the field-based protocol.

Table 5.1: Decision tree computation (seconds).

	Resources	SPDZ	MHM \mathbb{Z}_p	SHM \mathbb{Z}_{2^n}	BMR
Security level:		Malicious $t \leq n$	Malicious $t < n/2$	Semi-honest $t < n/2$	Malicious $t \leq n$
Online: time:	1 core	0.3005	3.0416	0.4641	0.5353
# rounds:		783	584	2746	28
Offline: time:	48 cores	5.2204	Not required	Not required	1041.8

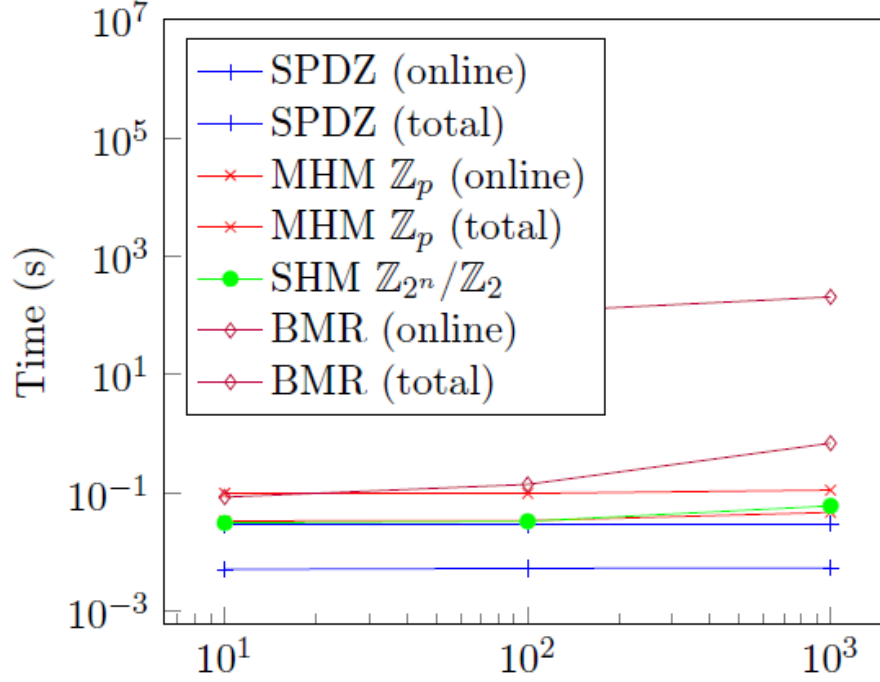


Figure 5.6: Benchmarking on Mean computation (X-axis=num. inputs)

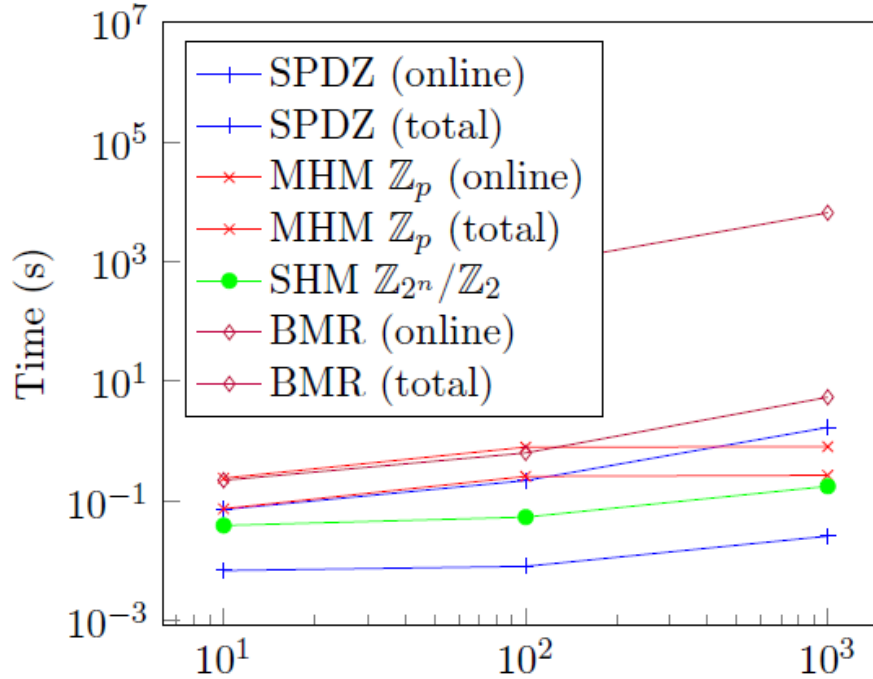


Figure 5.7: Benchmarking on Variance computation (X-axis=num. inputs)

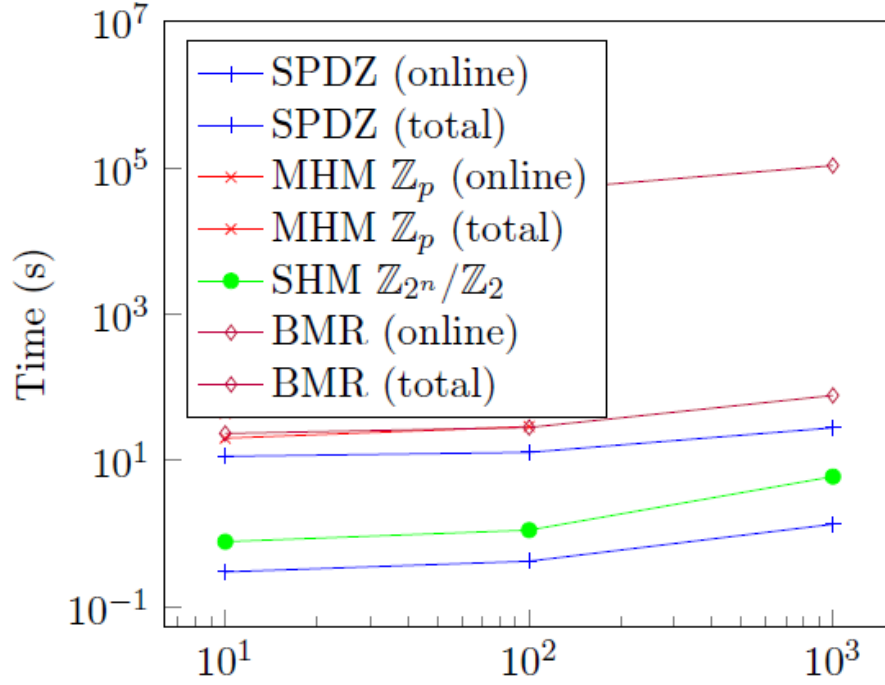


Figure 5.8: Benchmarking on US Census SQL query (X-axis=num. inputs)

Batch vectorization. We have implemented batch vectorization for the Ring-based protocol at the VM level. This works by defining the level of vectorization desired, and then the same single-execution code written at the compiler level is run on vectors of the specified length. For example, defining vectorization of level 64 for the decision tree inference problem means that inference is run on 64 inputs at the same time. This works by representing each element as a vector of 64, and running the MPC in parallel for each.

We ran these batch executions on the same problems as above; these results appear in Table 5.2. Observe that the “non-batch” and “Batch $\times 1$ ” both run a single execution, but there is a fixed overhead in the VM for running the batched experiments. Comparing these two columns, one can see that this overhead is quite high; we are working on reducing it. Beyond this, observe that the cost of batching many executions together is very minor. Thus, a single decision tree inference (without batching) takes approximately 0.5 seconds whereas 64 in parallel takes just under 6 seconds, or an average of under 0.1 second. We believe that by reducing the fixed

overhead, we will obtain that parallelism is essentially for free. This is of great importance in many real-world use cases where the same computation is carried out many times. For example, census statistics like the SQL query in our example would be computed for every state, and so could be vectorized.

Table 5.2: Running times for batch vectorization in seconds. $\text{Batch} \times N$ means running N executions in parallel (i.e., with vectors of length N).

	Non-batch	Batch $\times 1$	Batch $\times 8$	Batch $\times 32$	Batch $\times 64$
Mean (10 inputs)	0.031	0.139	0.138	0.136	0.149
Mean (100 inputs)	0.033	0.145	0.145	0.142	0.153
Mean (1000 inputs)	0.060	0.178	0.184	0.176	0.171
Variance (10 inputs)	0.039	0.362	0.371	0.391	0.381
Variance (100 inputs)	0.053	0.428	0.688	0.677	0.687
Variance (1000 inputs)	0.175	2.501	2.318	2.348	2.461
SQL (10 inputs)	0.779	10.233	10.335	10.997	11.285
SQL (100 inputs)	1.122	10.766	11.029	11.754	13.606
SQL (1000 inputs)	6.039	17.755	15.216	31.154	36.471
Decision tree	0.464	2.949	3.276	4.399	5.945

Open source. Our code is open source and available for free use. Our fork of SPDZ-2, including our extensions and hooks to them and changes to the compiler to support adding instructions and so on, can be found at <https://github.com/nec-mpc>. Furthermore, the extension required for plugging in the multi-party honest-majority protocol of [85] can be found at <https://github.com/cryptobiu/SPDZ-2-Extension-MpcHonestMajority>.

Future work This paper describes the first steps towards making the SPDZ compiler a general-purpose tool that can enable the use of MPC by software developers without MPC expertise. In order to complete this task, more work is needed in the following areas:

- *Efficiency:* The current run-time requires additional optimizations to achieve running-time that is comparable to that of a native protocol that works directly with a circuit and is optimized for latency or throughput. It is unreasonable to assume that a general compiler will achieve the same level of efficiency as a tailored optimized

version of a protocol. Nevertheless, the usability gains are significant enough so that a reasonable penalty (of say, 15%) is justified. An important goal is thus to achieve efficiency of this level, and we are currently working on this.

- *Protocol generality:* As we have argued, there is no single MPC protocol that is best for every task. On the contrary, we now understand that many different protocols of different types are needed for different settings. The best protocol depends on the efficiency goal (low latency or high throughput), the network setting (LAN or WAN), the function being computed (arithmetic or Boolean or mixed, and if mixed how many transitions are needed), and so on. In order to achieve this goal, more protocols need to be incorporated into the SPDZ compiler framework.

In addition to the above, we believe that an additional method should be added that outputs a circuit (arithmetic, Boolean or mixed) generated from the Python code. This deviates from the SPDZ run-time paradigm and requires running the protocol with a specific circuit, but it enables the use of the compiler in the more traditional circuit-compiler methodology that also has advantages. In particular, it can be used for protocols that have not been incorporated into the SPDZ run-time, and for optimized code that works specifically with a static circuit.

- *Compiler generality:* The SPDZ compiler is already very general and provides support for a rich high-level language. However, as more real use cases are discovered, it will need to be further enriched. This work is already being done independently on the original SPDZ compiler and we hope that these works will be merged, for the benefit of the general community.

Chapter 6 Application (2): 3-Party Computation for High-Level Functions

6.1 MPC for Bit Decomposition and Ring Composition

6.1.1 Introduction

As we have discussed above, the SPDZ compiler provides high-level algorithms for operations from numerical comparisons to fixed and floating point computations. These algorithms require the capability to decompose a basic element into its bit representation and back. Since the SPDZ protocol works over fields, it already contains these methods for field elements. However, it does *not* support bit decomposition and ring composition for ring elements. Since this is crucial for running SPDZ programs over ring-based MPC, in this section we describe a new method for bit decomposition and ring composition for the ring-based protocol of [6, 5]. We stress that our method works for any 3-party protocol based on replicated secret sharing as is the case for [6, 5], but it does not work for any ring-based protocol in general. We follow this strategy in order to achieve highly efficient bit decomposition and ring composition; since these operations are crucial and ubiquitous in advanced computations, making the operation as efficient as possible is extremely important.

Before beginning, we explain why bit decomposition and ring composition can be made much more efficient in the ring \mathbb{Z}_{2^n} . Consider the case of additive shares where the parties hold values s_i such that $\sum_{i=1}^n s_i = s$, where s is the secret. If the addition is in a field like \mathbb{Z}_p , then the values of all bits depend on all other bits. In particular, the value of the least significant bit depends also on the most significant bits; consider computing $16 + 8 \bmod 17$. The three least significant bits of 16 and 8 are zero, but the result is 7, which is 111 in binary. This is not the case in $GF[2^n]$ and bit decomposition is actually trivial in this field. However, since we typically use arithmetic circuits to embed numerical computations, we need integer addition and multiplication to be preserved in the field or ring. For this reason, the ring \mathbb{Z}_{2^n} has

many advantages. First, it allows for very efficient local operations. Second, the sum of additive shares has the property that each bit of the result depends only on the corresponding bit in each share, and the carry from the previous share. We will use this in an inherent way in order to obtain more efficient bit decomposition and ring composition protocols.

Contribution We propose new efficient MPC protocols for *bit decomposition* and *ring composition* operations, to convert a shared ring element to a series of shares of its bit representation and back, which is based on our MPC described in Chapter 3 and 4.

Efficient bit decomposition and ring composition are essential primitives for efficient MPC, since many real-world programs require both arithmetic computations, as well as comparison and other operations that require bit representation. However, such conversions are difficult to carry out, especially in the presence of malicious adversaries. This is due to the fact that malicious parties can change the values that they hold, and a secure protocol has to prevent such behavior. We overcome this by constructing protocols that are comprised of *only* standard ring-MPC operations (over shares of ring elements), standard bit-MPC operations (over shares of bits), and local transformations from *valid* ring-shares to *valid* bit-shares (and vice versa) that are carried out independently by each party. Since this is the case, the security is easily reduced to the security of the ring and bit protocols which have been proven.

Our bit-decomposition and ring-composition conversion protocols are constructed specifically for replicated secret sharing and between the ring \mathbb{Z}_{2^n} (for any n) and \mathbb{Z}_2 . Although this is a very specific scenario, it enables very high throughput secure computation of any functionality (in the setting of three parties, with at most one corrupted). In particular, the recent protocols of [6] and [5] can be used. These protocols achieve high throughput by requiring very low communication: in the protocol of [6] for semi-honest adversaries, each party sends a single bit (resp., ring element) per AND gate (resp., multiplication gate) when computing an arbitrary Boolean circuit (resp., arithmetic circuit over \mathbb{Z}_{2^n}). Furthermore, the protocol of [5] achieves security in the presence of malicious adversaries in this setting at the cost of just 7 times that of [6] (i.e., 7 bits/ring elements per AND/multiplication gate).

Our method utilizes local computations and native multiplications and additions in Boolean and ring protocols. As such, if the underlying Boolean and ring protocols

are secure for *malicious adversaries*, then the result is bit decomposition and ring composition that is secure for *malicious adversaries*. Likewise], if the underlying protocols are secure for semi-honest adversaries then so is the result.

6.1.2 Related Work

In the context of MPC, bit decomposition is originate from the work of Damgård et al. [39]. They showed the bit decomposition protocol for arbitrary linear secret sharing with *constant round* and $O(n \log n)$ communication (note that n is the size of field/ring). [39] also shows several protocol can be constructed from bit decomposition with constant round, and people recognized the power of free access to both of Boolean and arithmetic circuits. After that, [97] and [98] improve the order of communication complexity to (almost) linear.

One of the biggest drawback of previous bit decomposition is that, the size of the share for each bit in their bit decomposition is influenced by the size of original field. Namely, the share of each bit requires same size as the arithmetic value before bit decomposition, and thus the communication complexity is highly expensive. In contrast to this, Nishide and Ohta [99] propose several efficient MPC for useful functionalities while avoiding use of bit decomposition.

Ideally speaking, communication-efficient bit decomposition will be effective to perform a number of complex functionality efficiently. However, to reduce the communication of bit decomposition, we also need *modulus conversion* from shares for arithmetic values to shares for bits. However, bit decomposition which equipped with modulus conversion does not exist so far.

In addition, ring composition is considered as more complex than bit decomposition and efficient ring composition is not known. Therefore, the conversion between Boolean circuit and arithmetic circuit is limited and it obstruct to improve the efficiency of mixed circuit.

On the other hand, our proposed bit decomposition in this thesis involves modulus conversion. Therefore, we can utilize the power of bit decomposition with minimal communication overhead. Moreover, we also efficient ring composition which communication cost is almost same as bit decomposition. It realizes flexible conversion of Boolean/arithmetic circuit “feel free” and should support more efficient computation of complex circuit.

6.1.3 Communication-Efficient Bit Decomposition

Ring operations are extremely efficient for computing sums and products. However, in many cases, it is necessary to also carry out other operations, like comparison, floating point, and so on. In such cases, it is necessary to first *convert* the shares in the ring to shares of *bits*. For example, we can efficiently compute comparison (e.g., less-than) using a Boolean circuit, but we first need to hold the value in Boolean representation. This operation is called **bit decomposition**. Recall that a sharing of $x \in \mathbb{Z}_{2^n}$ is denoted by $[x]^{2^n}$ (and thus a sharing of a bit a is denoted by $[a]^2$). Writing $x = x^n \cdots x^1$ (as its bitwise representation with x^1 being the least significant bit), the bit decomposition operation is a protocol for converting a sharing $[x]^{2^n}$ of a single *ring element* $x \in \mathbb{Z}_{2^n}$ into n shares $[x^n]^2, \dots, [x^1]^2$ of its bit representation. We stress that it is not possible for each party to just locally decompose its shares into bits, because the addition of single bits results in a carry. To be concrete, assume that $x = 1101_2 = 13_{10} \in \mathbb{Z}_{2^4}$ (where subscript of 2 denotes binary, and a subscript of 10 denotes decimal representation). Then, an additive sharing of x could be $x_0 = 1011_2 = 11_{10}$, $x_1 = 1001_2 = 9_{10}$ and $x_2 = 1001_2 = 9_{10}$. If we look separately at each bit of x_0, x_1, x_2 , then we would obtain a sharing of $1011_2 = 11_{10} \neq x$ (this is computed by taking the XOR x_0, x_1, x_2).

Step 1 – local decomposition: In this step, the parties locally compute shares of the individual bits of their shares. Let the sharing $[x]^{2^n}$ be with values (x_0, x_2) , (x_1, x_0) and (x_2, x_1) . The parties begin by generating *shares of their shares* x_0, x_1, x_2 . This is a local operation defined by the following table:

Table 6.1: Reference for local re-sharing for bit-decomposition

	P_0	P_1	P_2
Original shares of \mathbf{x} :	(x_0, x_2)	(x_1, x_0)	(x_2, x_1)
New sharing of \mathbf{x}_0 :	$(x_0, 0)$	$(0, x_0)$	$(0, 0)$
New sharing of \mathbf{x}_1 :	$(0, 0)$	$(x_1, 0)$	$(0, x_1)$
New sharing of \mathbf{x}_2 :	$(0, x_2)$	$(0, 0)$	$(x_2, 0)$

Observe that each party can locally compute its sharing of the shares, without any interaction. In addition, each sharing is correct. The above local decomposition is actually carried out separately for *each bit* of the shares. Denote by x_i^j the j th bit of x_i

where 1 represents the least-significant bit; i.e., $x_i = (x_i^n, x_i^{n-1}, \dots, x_i^1) \in (\mathbb{Z}_2)^n$. Then, the j th bit of share x_0 is locally converted into the sharing $(x_0^j, 0), (0, x_0^j), (0, 0)$, the j th bit of share x_1 is locally converted into the sharing $(0, 0), (x_1^j, 0), (0, x_1^j)$, and the j th bit of share x_2 is locally converted into the sharing $(0, x_2^j), (0, 0), (x_2^j, 0)$. Observe that these are already shares of bits, and are thus actually $[x_0^j]^2, [x_1^j]^2, [x_2^j]^2$. At this point, the parties all hold shares of the bit representation of the shares. This is *not* a bitwise sharing of x , but just of x_0, x_1, x_2 . In order to convert these to a bitwise sharing of x , we need to add the shares. However, this addition must take into account the *carry*, and thus local share addition will not suffice.

Step 2 – add with carry: Our aim is to compute the bit representation of $x = x_0 + x_1 + x_2$ using the bitwise shares. Since this addition is modulo 2^n , we need to compute the carry. In the least significant bit, the required bit is just $[x^1]^2 = [x_0^1]^2 + [x_1^1]^2 + [x_2^1]^2 \bmod 2$ (i.e., using local addition of shares). However, we also need to compute the carry, which involves checking if there are at least two ones. This can be computed via the function $\text{majority}(a, b, c) = a \cdot b \oplus b \cdot c \oplus c \cdot a$ which requires 3 multiplications. Since this needs to be computed many times during the bit decomposition, it is important to reduce the number of multiplications. Fortunately, it is possible to compute majority with just a *single* multiplication by

$$\text{majority}(a, b, c) = (a \oplus c \oplus 1) \cdot (b \oplus c) \oplus b.$$

In order to see that this is correct, observe that

$$\begin{aligned} (a \oplus c \oplus 1) \cdot (b \oplus c) \oplus b &= a \cdot (b \oplus c) \oplus c \cdot (b \oplus c) \oplus (b \oplus c) \oplus b \\ &= a \cdot b \oplus a \cdot c \oplus b \cdot c \oplus c \cdot c \oplus b \oplus c \oplus b = a \cdot b \oplus a \cdot c \oplus b \cdot c. \end{aligned}$$

Having computed the carry, it is now possible to compute the next bit, which is the sum of $[x_0^2]^2, [x_1^2]^2, [x_2^2]^2$ and the carry from the previous bit. However, observe that since there are now *four* bits to be added, the carry can actually be *two bits*. This in turn means that *five* bits actually need to be added in order to compute the actual bit and to compute its two carries. Denote by c_j and d_j the carries computed from the j th bit. Then, we claim that the bit and its carries can be computed as follows. Compute $[\alpha_j]_1 = [x_1^j]^2 \oplus [x_2^j]^2 \oplus [x_3^j]^2$, $[\beta_j]^2 = \text{majority}(x_1^j, x_2^j, x_3^j)$ and

Observe that the last three columns equal the binary count of the number of ones in the first 5 columns (from 0 to 5), as required. Since the cost of computing **majority** is just a single multiplication, this means that the overall cost of the bit decomposition is *three multiplications* per bit (two majority computations and one multiplication for computing d_j).

We now show how to improve this to *two multiplications* per bit instead of three. The idea here is to not explicitly compute the two carry bits c_j, d_j , and instead to leave them implicit in the $\alpha_j, \beta_j, \gamma_j$ values. Specifically, we will show that $\beta_j + \alpha_j$ (with the sum over the integers) actually equals the sum of the carry.⁵

The actual computation is as follows. As above, compute $[\alpha_j]^2 = [x_1^j]^2 \oplus [x_2^j]^2 \oplus [x_3^j]^2$ and $[\beta_j]^2 = \text{majority}(x_0^j, x_1^j, x_2^j)$. However, differently to above, compute $[\gamma_j]^2 = \text{majority}(\alpha_j, \beta_{j-1}, \gamma_{j-1})$. Finally, compute $[x^j]^2 = [\alpha_j]^2 \oplus [\beta_{j-1}]^2 \oplus [\gamma_{j-1}]^2$.

We summarize the overall bit decomposition protocol in Protocol 6.1.

Protocol 6.1 : Communication-Efficient Bit Decomposition from \mathbb{Z}_{2^n} to $(\mathbb{Z}_2)^n$

- **Inputs:** Each party hold the share $[x]_i^{2^n}$ for a secret x , and the opcode `bit_decomp` for bit decomposition. Let $c_0 = d_0 = d_{-1} = 0$.
- **The protocol:**
 1. The parties perform local re-sharing for $[x]^{2^n}$ and obtain the shares $([x_0^n]^2, [x_0^{n-1}]^2, \dots, [x_0^1]^2)$, $([x_1^n]^2, [x_1^{n-1}]^2, \dots, [x_1^1]^2)$, and $([x_2^n]^2, [x_2^{n-1}]^2, \dots, [x_2^1]^2)$.
 2. For $j = 1, \dots, n$, the parties compute $[\alpha_j]^2 = [x_0^j]^2 \oplus [x_1^j]^2 \oplus [x_2^j]^2$, $[\beta_j]^2 = \text{majority}(x_0^j, x_1^j, x_2^j)$ and $[\gamma_j]^2 = \text{majority}(\alpha_j, c_{j-1}, d_{j-2})$. Then, compute $[x^j]^2 = [\alpha_j]^2 \oplus [c_{j-1}]^2 \oplus [d_{j-2}]^2$, $[c_j]^2 = [\beta_j]^2 \oplus [\gamma_j]^2$, and $[d_j]^2 = [\beta_j]^2 \cdot [\gamma_j]^2$.
 3. The party P_i output $(\text{bit_decomp}, ([x^n]_i^2, [x^{n-1}]_i^2, \dots, [x^1]_i^2))$ where $i \in \{0, 1, 2\}$.

This operation as the whole is denoted by $([x^n]^2, [x^{n-1}]^2, \dots, [x^1]^2) = \text{bit_decomp}([x]^{2^n})$.

Proof of correctness. We prove that this is correct by induction. The inductive

claim is that for every j , the bit x^j is the correct j th bit of the sum, and the value $\beta_j + \gamma_j \in \{0, 1, 2\}$ with the sum computed *over the integers*, is the carry from the sum $x_0^j + x_1^j + x_2^j + \beta_{j-1} + \gamma_{j-1}$. For $j = 1$, this is trivially the case, since $\beta_0 = \gamma_0 = 0$ and so the bit $x^1 = \alpha_1 = x_0^1 \oplus x_1^1 \oplus x_2^1$, and the carry is just $\text{majority}(x_0^1, x_1^1, x_2^1)$. Assume now that this holds for $j - 1$, and we prove for j . We prove the correctness of this inductive step via a truth table (as above, the computation is symmetric and so it only matter how many ones there are amongst x_0^j, x_1^j, x_2^j , and the value of $\beta_{j-1} + \gamma_{j-1}$).

Table 6.3: Truth table for checking correctness on x^j

x_0^j	x_1^j	x_2^j	β_{j-1}	γ_{j-1}	α_j	β_j	γ_j	x^j	Carry $\beta_j + \gamma_j$
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	0	1	0
1	1	0	0	0	0	1	0	0	1
1	1	1	0	0	1	1	0	1	1
0	0	0	1	0	0	0	0	1	0
1	0	0	1	0	1	0	1	0	1
1	1	0	1	0	0	1	0	1	1
1	1	1	1	0	1	1	1	0	2
0	0	0	1	1	0	0	1	0	1
1	0	0	1	1	1	0	1	1	1
1	1	0	1	1	0	1	1	0	2
1	1	1	1	1	1	1	1	1	2

In order to see that this is correct, observe that $x_0^j + x_1^j + x_2^j + \beta_{j-1} + \gamma_{j-1}$ should equal $x^j + 2 \cdot (\beta_j + \gamma_j)$, with all addition over the integers (note that the carry is multiplied by 2 since it is moved to the next bit). By observation, one can verify that this indeed holds for each row. We therefore conclude that the above method correctly computes the sum $[x^1]^2, \dots, [x^n]^2$ requiring *only two multiplications per bit*.

6.1.4 Communication-Efficient Ring Composition

In this section, we show how to compute $[x]_n$ from $[x^1]^2, \dots, [x^n]^2$, where $x = x^n \cdots x^1$ (or stated differently, where $x = \sum_{j=1}^n 2^{j-1} \cdot x^j$). At first sight, it may seem that it is possible for each party to simply locally compute $[x]^{2^n} = \sum_{j=1}^n 2^{j-1} \cdot [x^j]^2$, requiring

only local scalar multiplications and additions. However, this does not work since the shares $[x^j]^2$ are of bits and not of ring elements, and due to carries one cannot relate to each bit separately and naively embed the bits into ring elements.

We use a similar method to that of bit decomposition, but in reverse order. As in decomposition, there are two steps: local decomposition of the bit shares into three different shares, and then adding with carry. The difference here is that we need to *cancel* out the carry, rather than compute it as in decomposition. In order to see why, assume that we wish to compose two bit sharings into a single sharing in \mathbb{Z}_{2^2} . Let $x = 2$ be the value to be composed, and so the parties hold shares of 0 (for the least significant bit) and of 1. Assume now that the sharing of 0 is defined by $x_0^1 = 1$, $x_1^1 = 1$, and $x_2^1 = 0$ (and so $x_0^1 \oplus x_1^1 \oplus x_2^1 = 0$). Then, the sum in the ring of these three shares is actually 2. Thus, this value of 2 in the first bit needs to be *cancelled out* in the second bit. This is achieved by subtracting 1 (or XORing 1) from the second bit. To make this more clear, denote by $\text{bit}(x^j)$ the j th shared bit (as a bit), and by $\text{carry}(x^j)$ the integer carry of the integer-sum of the bit-shares of x^j . For example, if $x_0^j = 1$, $x_1^j = 1$ and $x_2^j = 0$ then $\text{bit}(x^j) = 0$ and $\text{carry}(x^j) = 1$ (the carry equals 1 and not 2, since we consider it as a carry and so it is moved to the left by one bit; stated differently, $x_0^j + x_1^j + x_2^j = \text{bit}(x^j) + 2 \cdot \text{carry}(x^j)$ where addition here is over the integers). Our protocol for ring composition works by having the parties in the j th step compute shares in the ring of the value $x^j = \text{bit}(x^j) + 2 \cdot \text{carry}(x^j) - \text{carry}(x^{j-1})$. Finally, they all locally compute $[x]^{2^n} = \sum_{j=1}^n 2^{j-1} \cdot [x^j]^{2^n}$. This is correct since

$$\begin{aligned}
\sum_{j=1}^n 2^{j-1} \cdot [x^j]^{2^n} &= \sum_{j=1}^n 2^{j-1} \cdot (\text{bit}(x^j) + 2 \cdot \text{carry}(x^j) - \text{carry}(x^{j-1})) \\
&= \sum_{j=1}^n 2^{j-1} \cdot \text{bit}(x^j) + \sum_{j=1}^n 2^{j-1} \cdot 2 \cdot \text{carry}(x^j) - \sum_{j=1}^n 2^{j-1} \cdot \text{carry}(x^{j-1}) \\
&= \sum_{j=1}^n 2^{j-1} \cdot \text{bit}(x^j) + \sum_{j=1}^n 2^j \cdot \text{carry}(x^j) - \sum_{j=0}^{n-1} 2^j \cdot \text{carry}(x^j) \\
&= [x]^{2^n} + 2^n \cdot \text{carry}(x^n) - \text{carry}(x^0) = [x]^{2^n}
\end{aligned}$$

where the third equality is by simply changing the range of the index j in the third term (from $1, \dots, n$ to $0, \dots, n-1$), and the last equality is due to the fact that in the ring \mathbb{Z}_{2^n} the carry to the $(n+1)$ th place is just ignored (and that $\text{carry}(x^0) = 0$).

We now describe the algorithm. Let $[x^1]^2, \dots, [x^n]^2$ be the input bitwise shares;

the output should be $[x]^{2^n} = \sum_{j=1}^n 2^{j-1} \cdot x^j$.

Step 1 – local decomposition: In this step, the parties locally compute shares of the individual bits of their shares, for each j . Specifically, as above, let the sharing $[x^j]^2$ be with values (x_0^j, x_2^j) , (x_1^j, x_0^j) and (x_2^j, x_1^j) . The parties generate *shares of their shares* as follows, via local computation only:

Table 6.4: Reference for local re-sharing for ring composition

	P_0	P_1	P_2
Original shares of x^j :	(x_0^j, x_2^j)	(x_1^j, x_0^j)	(x_2^j, x_1^j)
New sharing of x_0^j :	$(x_0^j, 0)$	$(0, x_0^j)$	$(0, 0)$
New sharing of x_1^j :	$(0, 0)$	$(x_1^j, 0)$	$(0, x_1^j)$
New sharing of x_2^j :	$(0, x_2^j)$	$(0, 0)$	$(x_2^j, 0)$

At this point, the parties hold $[x_0^j]^2, [x_1^j]^2, [x_2^j]^2$ for $j = 1, \dots, n$.

Step 2 – add while removing carry: For $j = 1, \dots, n$, the parties compute the shares $[\alpha_j]^2 = [x_0^j]^2 \oplus [x_1^j]^2 \oplus [x_2^j]^2$, $[\beta_j]^2 = \text{majority}(x_0^j, x_1^j, x_2^j)$ and $[\gamma_j]^2 = \text{majority}(\alpha_j, \beta_{j-1}, \gamma_{j-1})$, where $\beta_0 = \gamma_0 = 0$. (Recall that each majority computation requires one bit-wise multiplication.) Then, the j th bit of the result is mapped to a share $[x^j]^{2^n}$ of a ring element by computing

$$[v^j]^2 = [\alpha_j]^2 \oplus [\beta_{j-1}]^2 \oplus [\gamma_{j-1}]^2 \quad (10)$$

and *projecting* the result into the ring. That is, if a party holds a pair of bits $(0, 1)$ for its share of $[v^j]^2$, then it defines $[x^j]^2$ to simply be $(0, 1)$ in the ring \mathbb{Z}_{2^n} (i.e., the integers 0 and 1). Finally, the parties use local computation to obtain $[x]^{2^n} = \sum_{j=1}^n 2^{j-1} \cdot [x^j]^{2^n}$.

We stress that one should not confuse $[v^j]^2$ and $[x^j]^{2^n}$; they are both shares of the same value in some sense, *but actually define very different values*. To clarify this, observe that if $v_0^j = v_1^j = 1$ and $v_2^j = 0$, then (v_0^j, v_1^j, v_2^j) constitute a bit sharing of $[v^j]^2 = 0$. However, after projecting this into the ring, we have that it defines a ring-sharing of $[x^j]^{2^n} = 2$ (because $v_0^j + v_1^j + v_2^j = 0 \bmod 2$ but $v_0^j + v_1^j + v_2^j = 2 \bmod 2^n$).

We summarize the overall ring composition in Protocol 2.

Protocol 6.2 : Communication-Efficient Ring Composition from $(\mathbb{Z}_2)^n$ to \mathbb{Z}_{2^n}

- **Inputs:** Each party \mathcal{P}_j hold the share $[x^n]_i^2, [x^{n-1}]_i^2, \dots, [x^1]_i^2$ for a secret $x = \sum_{j=1}^n 2^{j-1} \cdot x^j$ and the opcode `ring_comp` for ring composition. Let $\beta_0 = \gamma_0 = 0$.
- **The protocol:**
 1. The parties perform local re-sharing for $[x^n]^2, \dots, [x^1]^2$ and obtain the shares $([x_0^n]^2, [x_0^{n-1}]^2, \dots, [x_0^1]^2)$, $([x_1^n]^2, [x_1^{n-1}]^2, \dots, [x_1^1]^2)$, and $([x_2^n]^2, [x_2^{n-1}]^2, \dots, [x_2^1]^2)$.
 2. For $j = 1, \dots, n$, the parties compute $[\alpha_j]^2 = [x_0^j]^2 \oplus [x_1^j]^2 \oplus [x_2^j]^2$, $[\beta_j]^2 = \text{majority}(x_0^j, x_1^j, x_2^j)$ and $[\gamma_j]^2 = \text{majority}(\alpha_j, \beta_{j-1}, \gamma_{j-1})$, and then compute $[v^j]^2 = [\alpha_j]^2 \oplus [\beta_{j-1}]^2 \oplus [\gamma_{j-1}]^2$. Then, the parties perform local re-sharing for $[v^j]^2$ as shown in Table 6.4 and obtain $[x_0^{j+1}]^2, [x_1^{j+1}]^2, [x_2^{j+1}]^2$.
 3. The parties P_i sets $[x]_i^{2^n} := ([v^n]_i^2 || [v^{n-1}]_i^2 || \dots || [v^1]_i^2)$ where $i \in \{0, 1, 2\}$. Finally, P_i outputs $(\text{ring_comp}, [x]_i^{2^n})$.

This operation as the whole is denoted by $[x]^{2^n} = \text{ring_comp}([x^n]^2, \dots, [x^1]^2)$.

Correctness: Correctness of the ring composition procedure is proven in a similar way to the decomposition.

6.1.5 Variants: Reducing the Round Complexity

It is possible to use known methods for adding in $\log n$ rounds, in order to reduce the round complexity of the bit decomposition. However, these come at a cost of much higher AND complexity. Instead, we utilize specific properties of our bit decomposition method in order to reduce the number of rounds, while only mildly raising the

number of ANDs. Our method is basically a variation of a *carry-select adder* [124], modified to be suited for bit decomposition. Observe that since the computation is essentially the same for bit decomposition and ring composition (regarding the computation of $\alpha_j, \beta_j, \gamma_j$), the same method here works for ring composition as well.

Recall that bit decomposition works by computing $[\alpha_j]^2 = [x_0^j]^2 \oplus [x_1^j]^2 \oplus [x_2^j]^2$, $[\beta_j]^2 = \text{majority}(x_0^j, x_1^j, x_2^j)$, and $[\gamma_j]^2 = \text{majority}(\alpha_j, \beta_{j-1}, \gamma_{j-1})$. The final shares are obtained by XORing these values and so does not add any additional rounds of communication. Observe that the α_j and β_j shares can all be computed in parallel in a single round. However, the γ_j values must be computed sequentially, since γ_j depends on γ_{j-1} . In order to explain the basic idea behind our tradeoff between computation and rounds, we show concretely how to reduce the number of rounds to approximately one half and one quarter, and then explain the general tradeoff:

Reducing to $n/2 + 2$ rounds. As described above, all of the α_j, β_j values can be computed in the first round, at the cost of exactly n AND gates. Next, the parties compute the following:

1. $\gamma_1, \dots, \gamma_{n/2}$ at the cost of $n/2$ rounds and $n/2$ AND gates,
2. $\gamma_{n/2+1}, \dots, \gamma_n$ under the *assumption* that $\gamma_{n/2} = 0$, at the cost of $n/2$ rounds and $n/2$ AND gates, and
3. $\gamma_{n/2+1}, \dots, \gamma_n$ under the *assumption* that $\gamma_{n/2} = 1$, at the cost of $n/2$ rounds and $n/2$ AND gates.

Observe that all three computations above can be carried out in parallel, and thus this requires $n/2$ rounds overall. Next, the parties use a MUX to compute which $\gamma_{n/2+1}, \dots, \gamma_n$ values to take; this is possible since $\gamma_{n/2}$ is already known at this point. This MUX uses a single AND gate per bit, coming to a total of $n/2$ AND gates, and a single round. The overall cost is $3n$ AND gates and $n/2 + 2$ rounds. Concretely for 32 bit values, this results in 96 AND gates and 17 rounds (instead of 64 AND gates and 32 rounds).

Reducing to $n/4 + 4$ rounds. This time we divide the γ_j values to be computed into 4 parts, as follows. In the first round, all α_j, β_j values are computed. Then:

1. $\gamma_1, \dots, \gamma_{n/4}$ are computed at the cost of $n/4$ rounds and $n/4$ AND gates,

2. In parallel to the above, $\gamma_{\frac{i \cdot n}{4}+1}, \dots, \gamma_{\frac{(i+1) \cdot n}{4}}$ for $i = 1, 2, 3$ are computed all in parallel, each under the *assumption* that $\gamma_{\frac{i \cdot n}{4}} = 0$ and that $\gamma_{\frac{i \cdot n}{4}} = 1$, at the cost of $n/4$ rounds and $n/4$ AND gates each (overall 6 such computations).

When all of the above are completed, the parties compute *sequentially* the MUX over $\gamma_{\frac{i \cdot n}{4}+1}, \dots, \gamma_{\frac{(i+1) \cdot n}{4}}$ given $\gamma_{\frac{i \cdot n}{4}}$ for $i = 1, 2, 3$ (each at a cost of $n/4$ AND gates and 1 round). The overall cost is $3.5n$ AND gates and $n/4 + 4$ rounds. Concretely for 32 bit values, this results in 112 AND gates and 12 rounds (instead of 64 AND gates and 32 rounds).

The general case. The above method can be used to divide the γ_j values into ℓ blocks. In this case, the number of rounds is $\frac{n}{\ell}$ to compute the γ values, and $\ell - 1$ to compute the sequential MUXes. Using a similar computation to above, we have that the overall number of rounds is $\frac{n}{\ell} + \ell$, and the number of AND gates is $n + \frac{(3\ell-2)n}{\ell}$. With this method, the number of rounds is minimized when $\frac{n}{\ell} = \ell$, which holds when $\ell = \sqrt{n}$ and results in $2\sqrt{n}$ rounds. In this case, the number of AND gates to be computed equals $4n - 2\sqrt{n}$. Importantly, this method provides a tradeoff between the number of rounds and the number of AND gates, since less blocks means less MUX computations. See Table 6.5 for a comparison on the number of rounds and AND gates, minimizing the number of rounds and minimizing the number of AND gates, when using our method. (Note that the minimum number of AND gates is always obtained by taking $\ell = 1$; i.e., by using the original method above.) These values are computed using the general equations above.

Table 6.5: Different parameters and their cost

Size n	Minimal ANDs		Minimal Rounds		
	ANDs	Rounds	ℓ	ANDs	Rounds
16	32	16	4	56	8
32	64	32	4	112	12
64	128	64	8	240	16
128	256	128	10	487	23

Somewhat surprisingly, it is possible to do even better by using a *variable-length carry-adder* approach. The idea behind this is that it is possible to start computing

the MUXes for the first blocks while still computing the γ values for the later blocks. To see why this is possible, consider the concrete example of $n = 16$ and $\ell = 4$. When dividing into equal-size blocks of length 4, the overall number of rounds is 8 (1 round for computing α_j, β_j and 7 for the rest). For this concrete case, we could divide the input into *five* blocks of sizes 2,2,3,4,5, respectively. Observe that the MUX needed using the result of the first block to choose between the two results of the second block can be computed in parallel to the last γ value on the third block. Likewise, the next MUX can be computed in parallel to the last γ value of the fourth block, and so on. In this way, there are no wasted rounds, and the overall number of rounds is reduced from 7 to 6. Although this is a modest improvement, for larger values of n , it is more significant. For example, we need 18 rounds for bit decomposition of 128-bit values, in contrast to 23 rounds with fixed-length blocks (see Table 6.5). We wrote a script to find the optimal division into blocks for this method, for various values of n ; the results appear in Table 6.6.

Table 6.6: Optimal block-size and costs for the variable-length approach (computation is from right-to-left)

Size n	ANDs	Rounds	Block Sizes
16	63	7	5, 4, 3, 2, 2
32	128	10	8, 7, 6, 5, 4, 2
64	255	13	11, 10, 9, 8, 7, 6, 5, 4, 4
128	519	18	16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 2

Observe that the number of ANDs required for this method is greater than in Table 6.5, thus further contributing to the aforementioned tradeoff. We stress that this tradeoff is significant since different parameters may provide better performance on slow, medium and fast networks.

Bit decomposition using conditional sum adders. We conclude with a different approach that is based on a conditional sum adder. This variant takes a divide-and-conquer approach to computing the blocks. That is, it splits the n -bit input into two blocks of $n/2$ bits, uses a conditional sum adder to compute the sum of the lower block with carry 0 and the sum of the higher block with carries 0 and 1, and then

uses MUX gates to select the correct outputs for the higher block. At the bottom level, a pair of bits is simply added using a full adder. This tree-based approach leads to a logarithmic number of rounds at an overall cost of $O(n \log n)$ AND gates, since there are a linear number of MUX gates at every level. The concrete costs for this method are presented in Table 6.7. As can be seen, the number of rounds is significantly reduced, but at the cost of a notable increase in the number of ANDs. In slow networks with fast computing devices, this approach can be preferable.

Table 6.7: Costs for the conditional-sum adder approach

Size n	ANDs	Rounds
16	121	6
32	280	7
64	631	8
128	1398	9

6.1.6 Security

Let v be a value. We say that $\text{type}(v) = \mathbb{Z}_{2^n}$ if $v \in \mathbb{Z}_{2^n}$ and we say that $\text{type}(v) = \mathbb{Z}_2$ if $v \in \{0, 1\}$. We will relate to the addition, scalar multiplication and multiplication of values below. In all cases, these operations are only possible for values of the *same type*.

In \mathcal{F}_{mpc} , we define a general MPC functionality that enables carrying out standard operations on shared values: addition, scalar multiplication and multiplication (beyond sharing input and getting output). However, in contrast to the usual definition, we define \mathcal{F}_{mpc} to carry out these operations on both shares of bits and shares of ring elements. In addition, the functionality enables the decomposition of a ring element in \mathbb{Z}_{2^n} to n shares of bits, and the ring composition of n shares of bits to a ring element. This provides a much more general functionality since computations can be carried out both using arithmetic circuits and Boolean circuits.

Functionality 6.1 : \mathcal{F}_{mpc} – The Mixed MPC Functionality

\mathcal{F}_{mpc} runs with parties P_1, \dots, P_m and the ring \mathbb{Z}_{2^n} , as follows:

- Upon receiving **(input, id, i, v)** from party P_i where $v \in \mathbb{Z}_{2^n}$ or $v \in \{0, 1\}$ and id has not been used before, \mathcal{F}_{mpc} sends **(input, id, i)** to all parties and locally stores (id, v) .
 - Upon receiving **(add, id_1, id_2, id_3)** from all *honest* parties, if there exist v_1, v_2 such that (id_1, v_1) and (id_2, v_2) have been stored and $\text{type}(v_1) = \text{type}(v_2)$, and id_3 has not been used before, then \mathcal{F}_{mpc} locally stores $(id_3, v_1 + v_2)$.
 - Upon receiving **(scalarmult, id_1, id_2, c)** from all *honest* parties, if there exists a v such that (id_1, v) has been stored and $\text{type}(c) = \text{type}(v)$, and id_2 has not been used before, then \mathcal{F}_{mpc} locally stores $(id_2, c \cdot v)$.
 - Upon receiving **(mult, id_1, id_2, id_3)** from all parties, if there exist v_1, v_2 such that (id_1, v_1) and (id_2, v_2) have been stored and $\text{type}(v_1) = \text{type}(v_2)$, and id_3 has not been used before, then \mathcal{F}_{mpc} locally stores $(id_3, v_1 \cdot v_2)$.
 - Upon receiving **(decompose, id, id_1, \dots, id_n)** from all parties, if there exists a v such that (id, v) has been stored and $\text{type}(v) = \mathbb{Z}_{2^n}$, and id_1, \dots, id_n have not been used before, then \mathcal{F}_{mpc} locally stores (id_i, v_i) for $i = 1, \dots, n$, where $v = v_1, \dots, v_n$.
 - Upon receiving **(recompose, id_1, \dots, id_n, id)** from all parties, if there exist v_1, \dots, v_n such that (id_i, v_i) has been stored and $\text{type}(v_i) = \mathbb{Z}_2$ for all $i = 1, \dots, n$, and id has not been used before, then \mathcal{F}_{mpc} locally stores (id, v) , where $v = v_1, \dots, v_n$.
 - Upon receiving **(output, id, i)** from all parties, if there exists a v such that (id, v) has been stored then \mathcal{F}_{mpc} sends **(output, id, v)** to party P_i .
-

The multiplication protocol. A formal description of the protocol that securely computes the multiplication functionality $\mathcal{F}_{\text{MULT}}$ in the \mathcal{F}_{CR} -hybrid model appears in Protocol 7.

Observe that **add** and **scalarmult** in \mathcal{F}_{mpc} are operations that depend only on the honest parties. This is due to the fact that they involve local operations only, and

thus the adversary cannot interfere in their computation. The standard \mathcal{F}_{mpc} functionality fulfilled by secret-sharing based protocols is the same as Functionality 6.1, with the exception that all operations are of one type only and there are not **decompose** or **recompose** operations. We denote the standard \mathcal{F}_{mpc} functionality that works over the ring R by $\mathcal{F}_{\text{mpc}}^R$ (and so denote $\mathcal{F}_{\text{mpc}}^{\mathbb{Z}_2}$ for bits and $\mathcal{F}_{\text{mpc}}^{\mathbb{Z}_{2^n}}$ for the ring \mathbb{Z}_{2^n}).

Security of bit decomposition and ring composition. The fact that our protocols are secure follow immediately from the fact that they are comprised solely of the following elements:

- Local transformation operations from *valid* bit shares to *valid* ring shares and vice versa,
- Bit-MPC add and multiply operations over bit shares, and
- Ring-MPC add and multiply operations over ring shares.

Since the MPC operations use secure protocols and work on valid shares of the appropriate type, these operations are securely carried out. Furthermore, since the local transformations require no communication, an adversary cannot cheat. Thus, the combination of the bit and ring protocols, along with the bit decomposition and ring composition protocols presented above, constitute a protocol that securely computes the mixed MPC functionality \mathcal{F}_{mpc} .

In order for the above to work, we need the bit and ring protocols to have the property that the original simulator (that did not consider bit-decomposition) also successfully simulate the protocol in the presence of the bit-decomposition protocols. It is straightforward to see that the simulation of the new protocol is exactly the same as before except for outputs that depend on a share of some value. This happens when local bit decomposition converts a share into a secret shared value. This simulation is not trivial since the above MPC functionality does not keep shares as its internal state. Nevertheless, fortunately, this exception does not happen in our functionality since full bit-decomposition does not reveal any value that depends on a share. By being deliberate in this point, we are able to simulate the functionality as before.

6.1.7 Efficiency

The complexity of our MPC conversions of shares between that of \mathbb{Z}_{2^n} and that of \mathbb{Z}_2^n are given in Table 6.8. These numbers refer to the three-party protocol of [5]

that is secure in the presence of malicious adversaries. The most optimized version of that protocol requires each party to send just 7 bits per AND gate, meaning an overall cost of 21 bits per AND gate for all parties. In Table 6.8 we provide the communication cost and number of rounds for our protocol, with different tradeoffs between computation and round complexity:

Table 6.8: Complexity of decomposition and ring composition

Version	Total comm. bits	Rounds
Basic conversion ($n = 32$)	1,344	32
Variable-length adder ($n = 32$)	2,688	10
Conditional-sum adder ($n = 32$)	5,880	7
Basic conversion ($n = 128$)	5,376	128
Variable-length adder ($n = 128$)	10,899	18
Conditional-sum adder ($n = 128$)	29,358	9

We now compare our protocol to the previous best protocols. We stress that previous protocols work generically for *any* ring, and as such are more general. However, this shows that much can be gained by focusing on rings of specific interest, especially the ring of integers which is of interest in many real-world computations. In Table 6.9, we present the cost of our protocols versus those of [39], [99] and [119], when applied to the ring $\mathbb{Z}_{2^{32}}$. In all cases, we consider the concrete cost when using the low-communication three-party protocol of [5] that requires only 7-bits of communication per party per AND gate. The results show the striking improvement our method makes over previous protocols, for the *specific case* of the ring \mathbb{Z}_{2^n} , and when using replicated secret sharing. For our protocol, we present the costs for the 32-round version, with minimum AND complexity.

Table 6.9: Comparison of Complexity of 32-bit Integer Conversions for Secure 3-Party Computation

Protocol	Method	Total comm. bits	Rounds
Bit Decomposition	[39] + [5]	723,912	38
	[99] + [5]	408,072	25
	Ours + [5]	1,302	32
Ring-Composition	[119] + [5]	340,704	62
	Ours + [5]	1,302	32

We remark that a direct conversion of the SPDZ bit decomposition method to the case of rings would yield the complexity of [39]+[5] in Table 6.9. Thus, our special-purpose conversion protocols are significant with respect to the efficiency of the result.

6.2 MPC for Exponentiation

6.2.1 Introduction

An Use Case: Securing Key Management by MPCs Basically, when we want to process confidential information on distributed systems, MPC could be an answer. As one of notable example of such applications, we can consider distributed ledgers for cryptocurrency, as we know Blockchain. The protection of secret keys in cryptosystems is an critical issue in several systems, in particular distributed system. Since the authority managing secret keys could be a single point of failure, key management is a problem that plague system engineers. However, even today, the signing key of the Blockchain is often managed by depositing with trusted authority such as exchanges in many services, thus it cannot be said that it is managed securely enough.

One of the solutions against this issue, we can consider applying MPCs to compute digital signatures among the nodes of distributed ledger, while concealing its signing keys. These research are also probably best known as distributed signatures or threshold signatures [122, 57, 84]. The secret sharing-based MPCs like [6, 5, 34, 92] can construct distributed signature schemes, since these MPCs

can compute any functions by composition of primitive MPC operations.

Difficulty To handle cryptographic operation like digital signatures, we should design MPC protocols for algebraic operations. In particular, modular exponentiation is most important building block for constructing many cryptographic protocols. Of course, the modular exponentiation is considered as the instruction widely-used other than cryptography, and it should be provided as a basic instruction.

However, we know that the cryptographic operations basically require large computation cost. Therefore, MPCs for cryptographic operations should be more inefficient.

Secret sharing-based MPCs generally requires communication between parties for computing multiplication. Hence, the complexity of the MPCs are dominated by the total number and the depth of the multiplications. The number of multiplications indicates the communication complexity of the MPC and the depth of multiplications indicates the round complexity (i.e., the number of communications) of the MPC.

One more important things is that non-obviousness of type conversion on MPCs. For non-MPC computations, we can easily perform both of arithmetic operations (e.g., addition, multiplication) and bit-wise operation (e.g., XOR, AND, left/right-shift). On the other hand, in MPCs, the data for arithmetic operation should be shared by arithmetic shares and the data for boolean operation should be shared by boolean shares. The conversion between these distinct type of shares also requires MPCs, namely additional communication and computation cost. A general method for modular exponentiation as known as square-and-multiply (or binary exponentiation) also needs to deal with boolean and arithmetic operations.

The MPCs based on replicated secret sharing are generally efficient than other frameworks, but the cryptographic operations are still heavy even for these schemes, let alone modular exponentiation as mentioned above. In this paper, as an important tool to apply MPC to distributed signatures, we focus on how to construct MPC for modular exponentiation on recent MPC frameworks.

Contribution In this section, we propose a new MPC protocol for modular exponentiation with public base for MPC framework based on replicated secret sharing [6, 5, 34, 92]. These frameworks are known as best practice for 3-party computation, but still unsuitable for cryptographic operation since these operations require much amount of multiplications.

Our proposed scheme is dedicated for modular exponentiation, which is based on the structure of secret sharing deployed by the frameworks of [6, 5, 34, 92].

More precisely, previous MPCs for modular exponentiation based on replicated secret sharing require $O(n^2)$ communication complexity by processing square-and-multiply method on MPC, where n is the size of the secret in bits. On the other hand, the proposed schemes in this paper require $O(n)$ communication without deteriorating the order of round complexity. For more concrete comparison, see Sect. 6.3.2.

We will show three types of construction, depending on the size of the modulus. First is the case when the modulus is power of 2, and second is the case for modulus is prime. Third scheme is also the case when the modulus is prime, with additional condition about the base and the exponent.

In addition, as an application of the proposed scheme, we consider applying this protocols to the distributed signatures, which generates signatures while concealing signing keys. We will show the experimental results assuming a scenario of distributed signatures.

6.2.2 Related Work

Before describing the proposed scheme, we will see the standard way to compute exponentiation on MPC. As the best of our knowledge, there is no dedicated MPC protocol for computing exponentiation

Protocol 6.3 : Previous work: modular exponentiation with public base

- **Inputs:** Each party P_i holds a public value a , the share $[x]_i^q$ for a secret x and the opcode `pub_expo` for public exponentiation.
- **The protocol:**
 1. The parties perform $[x_{n-1}]_i^2, \dots, [x_0]_i^2 = \text{bit_decomp}([x]_i^q)$
 2. For $j = 0, \dots, n-1 : [x_j]_i^q = \text{inject}([x_j]_i^2)$
 3. $[y]_i^q = [1]_i^q$
 4. For $j = n-1, \dots, 0 : [y]_i^q = a^{2^j} \cdot [x_j]_i^q \cdot [y]_i^q + (1 - [x_j]_i^q \cdot [y]_i^q)$
 5. The parties output (`pub_expo`, $[y]_i^q$)

This operation as the whole is denoted by $[y]^q = \text{pub_expo}(a, [x]^q)$

Protocol 6.3 is an implementation of the "square-and-multiply" method, which is a standard way of computing modular exponentiation. This method can be applied also the base is private.

Efficiency We can easily see that the dominant part of complexity in this procedure is the for-loop. This protocol takes $R_{bd} + R_{inj} + 2n$ round complexity and $C_{bd} + C_{inj} + 3n^2$ communication complexity, where C_{bd}, C_{inj} are the communication complexity of bit-decomposition and bit-injection, and R_{bd}, R_{inj} are round complexity of bit-decomposition and bit-injection, respectively. If we apply the scheme of [6, 7], Protocol 6.3 takes $3n + 2$ rounds and $15n^2 + 6n$ -bit communication complexity.

6.2.3 A Key Technique: Skew Exponentiation

Our basic idea is, like bit-decomposition of [7, 92, 77], to compute the shares of $a^{x_1}, a^{x_2}, a^{x_3}$ for some base a and share $[x = x_1 + x_2 + x_3]^q$. If we have such values, we can easily see that the exponentiation can be computed by $a^x = a^{x_1} \cdot a^{x_2} \cdot a^{x_3}$.

First, we introduce a new skew operation, named "Skew Exponentiation" as a first step of proposed scheme. After that, we will describe how to apply it to MPC exponentiation.

Looking back on the procedure of `skew_decomp`, what make this trick possible is the structure of replicated secret sharing. More precisely, in 2-out-of-3 replicated secret sharing, all values are retained in either two parties. Therefore, the parties can obtain a 2-out-of-3 share of each sub-shares x_i ($i \in \{0, 1, 2\}$) without communication by considering the value that they don't own as 0.

In this construction method, each element consisting $[x_i]^q$ is x_i itself or 0. Moreover, in the problem setting of modular exponentiation with public base, the base a is known. Hence, the parties that have x_i can also compute a^{x_i} directly. From this discussion, we can construct skew exponentiation as follows.

Protocol 6.4 : Skew Exponentiation

- **Inputs:** Each party P_i holds a public value a , the share $[x]_i^q$ for a secret x and the opcode `skew_expo` for public exponentiation.
- **The protocol:**
 1. Each party P_i sets
 - Let $[a^{x_0}]^q = ((a^{x_0}, a^{x_0}), (a^{x_0}, 0), (0, 0))$
 - Let $[a^{x_1}]^q = ((0, 0), (a^{x_1}, a^{x_1}), (a^{x_1}, 0))$
 - Let $[a^{x_2}]^q = ((a^{x_2}, 0), (0, 0), (a^{x_2}, a^{x_2}))$
 - Output (`skew_expo`, $[a^{x_0}]^q$, $[a^{x_1}]^q$, $[a^{x_2}]^q$)
 2. The parties output (`skew_expo`, $([a^{x_0}]_i^q, [a^{x_1}]_i^q, [a^{x_2}]_i^q)$).

This operation as the whole is denoted by $([a^{x_0}]^q, [a^{x_1}]^q, [a^{x_2}]^q) = \text{skew_expo}(a, [x]^q)$

6.2.4 Communication-Efficient Modular Exponentiation

We recall that what we want to compute is $a^x = a^{x_0+x_1+x_2 \bmod m}$. Note that x is *not* equal to $x_0 + x_1 + x_2$ but $x_0 + x_1 + x_2 \bmod m$. Therefore, we couldn't compute the share $[a^x]$ by $[a^{x_0}] \cdot [a^{x_1}] \cdot [a^{x_2}]$ naively, without checking whether the sum $x_0 + x_1 + x_2$ goes over modulus.

Scheme 1: the case when modulus is prime

Next we consider the case the modulus is prime p .

In this case, we can see that $a^x = a^{x'+kp} = a^x a^k$ where $k \in \{0, 1, 2\}$ by Fermat's little theorem. Namely, $x_0 + x_1 + x_2$ can go over the modulus at most twice. Therefore, we should check modulus overflow at the point of computing $x_0 + x_1$ and $(x_0 + x_1) + x_2$. If the these values go over the modulus p , we can fix it by multiplying a^{-1} .

To detecting the modulus overflow, we use bit-decomposition in this protocol. The point is that, if a certain value a is larger than the modulus p , the parity of $a \bmod p$ is flipped, since p is *odd*. Therefore, when we check the least significant bit of x_0, x_1 , and $x_0 + x_1 \bmod p$, if the parity is not consistent, it means that $x_1 + x_2$ exceed p .

We show the first our modular exponentiation protocol in Protocol 6.5. Step 1 is the skew exponentiation shown in 6.4, and Step 2 is “temporal” result of the modular exponentiation. As mentioned the above, we should check whether the exponent of the result in Step 2 overflow the modulus, and fix if it indeed overflow. Step 3–6 is the description of modulus overflow check for $x_0 + x_1$, and similarly Step 7–9 is check for $(x_0 + x_1) + x_2$. What we actually need are only least significant bit (LSB) of these values, we don't have to compute full procedure of `bit_decomp`, but can close the process when we get LSBs of the values.

Efficiency: Each `bit_decomp` and multiplication can be performed in parallel. In the above procedure, Step 2 takes 2-round and $6n$ -bit communication. Each `bit_decomp` takes $(n+1)$ -round and $10n+4$ -bit communication (using [77] since q is prime) and Step 3, 4, 5, 7, and 8 can be done in parallel. Steps 10 and 11 take 2-round and $6n$ -bit communication respectively and these steps can be done in parallel. Step 12 and 13 take 1-round and $6n$ -bit communication respectively. In total, `modexpp` takes $2 + (n+1) + 2 + 1 + 1 = (n+7)$ -round and $6n + 5 \cdot (10n+4) + 2 \cdot 6n + 2 \cdot 6n = (80n+20)$ -bit communication complexity.

Protocol 6.5 : Modular Exponentiation with prime modulus

- **Inputs:** Each party P_i holds a public value a , the share $[x]_i^q$ for a secret x and the opcode modexp_p for modular exponentiation with prime modulus.

- **The protocol:**

1. $([a^{x_0}]^q, [a^{x_1}]^q, [a^{x_2}]^q) \leftarrow \text{skew_expo}([x]^q)$
2. $[s]^q = [a^{x_0}]^q \cdot [a^{x_1}]^q \cdot [a^{x_2}]^q$
3. $[d_{n-1}]^2, \dots, [d_0]^2 = \text{bit_decomp}([x_0]^q)$
4. $[e_{n-1}]^2, \dots, [e_0]^2 = \text{bit_decomp}([x_1]^q)$
5. $[f_{n-1}]^2, \dots, [f_0]^2 = \text{bit_decomp}([x_0 + x_1]^q)$
6. $[b_1]^2 = [x_0 + x_1 > p]^2 = [d_0 \oplus e_0 \neq f_0]^q = [d_0 \oplus e_0 \oplus f_0]^q$
7. $[g_{n-1}]^2, \dots, [g_0]^2 = \text{bit_decomp}([x_2]^q)$
8. $[h_{n-1}]^2, \dots, [h_0]^2 = \text{bit_decomp}([x_0 + x_1 + x_2]^q)$
9. $[b_2]^2 = [x_0 + x_1 + x_2 > p]^2 = [f_0 \oplus g_0 \neq h_0]^q = [f_0 \oplus g_0 \oplus h_0]^q$
10. $[b_1]^q = \text{bit_inject}([b_1]^2)$
11. $[b_2]^q = \text{bit_inject}([b_2]^2)$
12. $[t]^q = [s]^q \cdot [b_1]^q \cdot a^{-1} + [s]^q(1 - [b_1]^q)$
13. $[t]^q = [t]^q \cdot [b_2]^q \cdot a^{-1} + [t]^q(1 - [b_2]^q)$
14. The parties output $(\text{modexp}_p, [t]^q)$

This operation as the whole is denoted by $([t]^q) = \text{modexp}_p(a, [x]^q)$

Scheme 2: the case where modulus is power of 2

Next we consider the case when $q = 2^n$ for some $n \in \mathbb{Z}$. To consider this case, we recall Euler's theorem.

Theorem 6.2.1 (Euler's theorem). *If n and a are coprime positive integers, $a^{\phi(n)} = 1 \pmod n$ where $\phi(\cdot)$ is Euler's totient-function.*

By Euler's theorem, if a is prime, $a^{2^{n-1}} = 1 \pmod{2^n}$, which implies $a^{2^n} = 1$

mod 2^n . Namely, in this case, we don't have to check the overflow of exponent.

Protocol 6.6 : Modular Exponentiation with the case where modulus is power of 2

- **Inputs:** Each party P_i holds a public value a , the share $[x]_i^q$ for a secret x and the opcode `modexp2n` for modular exponentiation with the modulus of power of 2.
- **The protocol:**
 1. $[a^{x_0}]^q, [a^{x_1}]^q, [a^{x_2}] = \text{skew_expo}([x]^q)$
 2. $[s]^q = [a^{x_0}]^q \cdot [a^{x_1}]^q \cdot [a^{x_2}]^q$
 3. output (`modexp2n`, $[s]^q$).

This operation as the whole is denoted by $([s]^q) = \text{modexp}_{2n}(a, [x]^q)$

Note that we cannot apply this procedure if a is even. In addition, it is difficult to apply the technique like Scheme 1 since there is no multiplicative inverse for all even value in \mathbb{Z}_{2^n} , that is we cannot compute a^{-1} on \mathbb{Z}_{2^n} if a is even. However, if we encounter case to apply the Scheme 2, this is very efficient.

Efficiency Scheme 2 requires requires only 2 multiplication for n -bit elements and *no bit_decomp*. Total cost of Scheme 2 is 2 rounds and $6n$ -bit communication complexity.

Scheme 3: special case that the discrete logarithm is small

We can consider the case when the size of the base and the exponent value are different. For example, we consider the case when $p = 2q + 1$, and $x_0, x_1, x_2 \in \mathbb{Z}_q$, $a \in \mathbb{Z}_p$. In such case, $x_0 + x_1 + x_2$ can exceed p at most *once*.

Protocol 6.7 : Modular Exponentiation in the special case where $p = 2q + 1$

- **Inputs:** Each party P_i holds a public value a , the share $[x]_i^q$ for a secret x and the opcode `modexp_sp` for modular exponentiation in the special case.

- **The protocol:**

1. $[a^{x_0}]^p, [a^{x_1}]^p, [a^{x_2}]^p = \text{skew_expo}([x]^q)$
2. $[s]^p = [a^{x_0}]^p \cdot [a^{x_1}]^p \cdot [a^{x_2}]^p$
3. $[d_{n-1}]^2, \dots, [d_0]^2 = \text{bit_decomp}([x_0 + x_1]^q)$
4. $[e_{n-1}]^2, \dots, [e_0]^2 = \text{bit_decomp}([x_1]^p)$
5. $[f_{n-1}]^2, \dots, [f_0]^2 = \text{bit_decomp}([x_0 + x_1 + x_2]^p)$
6. $[b]^2 = [x_0 + x_1 + x_2 > p]^2 = [d_0 \oplus e_0 \oplus f_0]^2$
7. $[b]^p = \text{bit_inject}([b]^2)$
8. $[t]^p = [s]^p \cdot [b]^p \cdot a^{-1} + [s]^p(1 - [b]^p)$
9. output (`modexp_sp`, $[s]^q$).

This operation as the whole is denoted by $([s]^q) = \text{modexp_sp}(a, [x]^q)$

In addition, if the case $p > 3q + 1$, we don't have to check the overflow of $x_0 + x_1 + x_2$ since $x_0 + x_1 + x_2 \bmod p = x_0 + x_1 + x_2$ in this parameter.

Protocol 6.8 : Modular Exponentiation in the special case where $p > 3q + 1$

- **Inputs:** Each party P_i holds a public value a , the share $[x]_i^q$ for a secret x and the opcode `modexp_sp2` for modular exponentiation in the special case.

- **The protocol:**

1. $[a^{x_0}]^p, [a^{x_1}]^p, [a^{x_2}]^p = \text{skew_expo}([x]^p)$
2. $[s]^p = [a^{x_0}]^p \cdot [a^{x_1}]^p \cdot [a^{x_2}]^p$
3. output (`modexp_sp2`, $[s]^p$).

This operation as the whole is denoted by $([s]^q) = \text{modexp_sp2}(a, [x]^q)$

Efficiency: Scheme 3 requires less number of invocation of `bit_decomp`. In particular, in the case where $3q+1 \leq p$, we can perform same procedure as Scheme 2. If the case where $2q+1 < p \leq 3q+1$ total cost of Scheme 3 is obviously $n+4$ rounds and $(36n-18)$ -bit communication complexity. If the case where $3q+1 < p$, Scheme 3 takes only 2 rounds and $6n$ -bit communication as same as Scheme 2.

6.2.5 Security

In this section, we discuss the proof of the security for the proposed scheme described in Sect. 6.2.4.

Universal Composability First, we confirm how the security of sub-protocols (like addition and multiplication described in Sect. 3.3.3, and local-resharing of sub-shares) can imply the security of the proposed scheme.

Here we describe the concept of *universal composability* (UC) framework proposed by [29]. Protocols which is secure in UC framework maintain its security even if it is composed with arbitrary other (secure and insecure) protocols. In particular, [78] clarified a condition under which the security of protocols implies the security of these protocols under universal composition as follows.

Proposition 6.2.2 (Thm. 1.5 in [78]). *Every protocol that is secure in the stand-alone model and has start synchronization and a straight-line black-box simulator is secure under concurrent general composition (universal composition).*

In the above theorem, “straight-line” simulator means that the non-rewinding simulator, and “start synchronization” means that the inputs of all parties are fixed before the execution begins (also called as “input availability”).

Our protocols and sub-protocols in this paper satisfies start synchronization. Therefore, it is sufficient to prove security in the classic stand-alone setting and automatically derive universal composability.

Security of Sub-protocols The security of sub-protocols described in Sect. 3.3.3 are proven in [6].

The proof in [6] consist of three steps as follows. Here we denote $\pi^{\mathcal{F}} \equiv f$ to say that π privately computes f in the \mathcal{F} -hybrid model.

1. Proving the sub-protocols π privately compute f in the \mathcal{F}_{mult} -hybrid model in the presence of one semi-honest corrupted party, where \mathcal{F}_{mult} is an ideal functionality for computing multiplication (namely, $\pi^{\mathcal{F}_{mult}} \equiv f$).

2. Proving a protocol ρ privately computes \mathcal{F}_{mult} in the \mathcal{F}_{CR} -hybrid model in the presence of one semi-honest corrupted party, where \mathcal{F}_{CR} is an ideal functionality for computing correlated randomness (namely, $\mathcal{F}_{mult} \equiv \rho^{\mathcal{F}_{CR}}$).
3. Proving a protocol σ privately computes \mathcal{F}_{CR} in plain model in the presence of one semi-honest corrupted party (namely, $\sigma \equiv \mathcal{F}_{CR}$).

The above three steps and the composition theorem described in Theore 6.2.2 lead $\pi^{\rho^\sigma} \equiv f$, which concludes the proof.

Security of Our Protocols Adding to the sub-protocols in Sect. 3.3.3, our protocols contain one more sub-protocol, that is, local exponentiation (or local re-sharing of sub-shares).

Fortunately, we can easily confirm that step 1 of the above proof still can be proven even π contains local re-sharing of sub-shares since this functionality consist of local computation only, as same as addition protocol. This is not affect the simulation in the \mathcal{F}_{mult} -hybrid model and $\pi^{\mathcal{F}_{mult}} \equiv f$. Regarding Step 2 and 3 of the proof, we can apply same proof as [6] for our protocol since we adopt same algorithms for multiplication and correlated randomness.

Finally, we can apply the proof of 6.2.5 to our protocol and thus all sub-protocols including local re-sharing of sub-shares are secure in terms of Definition 2.4.2. As described above, theorem 6.2.2 [78] guarantee that our all sub-protocols are secure in UC model. Therefore, by the UC composition theorem [29], we can prove the security of our protocols in Sect. 6.2.4.

On the Security for Malicious Adversaries We recall that our protocols in this paper are basically secure against *semi-honest* adversaries (see Definition 2.4.2). However, the protocols also can be secure in the presence of malicious adversaries by applying the technique of [56] (or its optimized version [5]), which allows us to construct a 3PC for malicious adversaries from 3PC protocols for semi-honest adversaries. If we apply the scheme in [5], the communication complexity is roughly 7 times that of the protocols in Sect 6.2.4.

6.2.6 Efficiency

We summarize the round and communication complexity for each protocol in Table 6.10. Basically, our proposed schemes requires $O(n)$ round and $O(n)$ -bit communication complexity, whereas previous scheme requires $O(n^2)$ -bit communication.

As an example, we show the comparison of communication bits between previous scheme and Scheme 1. We can clearly see how efficient the proposed schemes is compared with the previous scheme. The previous scheme takes over 300kb when $n=256$ bit. On the other hand, Scheme 1 requires 15,360-bit communication. As for Scheme 2 or 3 with $3q + 1 \leq p$ case, these takes only 1,536-bit communication.

Table 6.10: Complexity of MPC for Modular Exponentiation over Replicated Secret Sharing

Method	Round	Communication
Previous (Square-and-Multiply)	$2\lceil \log q \rceil + 1$	$12\lceil \log q \rceil^2 + 6\lceil \log q \rceil - 6$
Scheme 1 (q is prime)	$\lceil \log q \rceil + 7$	$80\lceil \log q \rceil - 20$
Scheme 2 (q is power of 2)	2	$6\lceil \log q \rceil$
Scheme 3 with $2q + 1 < p \leq 3q + 1$	$\lceil \log p \rceil + 4$	$36\lceil \log p \rceil - 18$
Scheme 3 with $3q + 1 \leq p$	2	$6\lceil \log p \rceil$

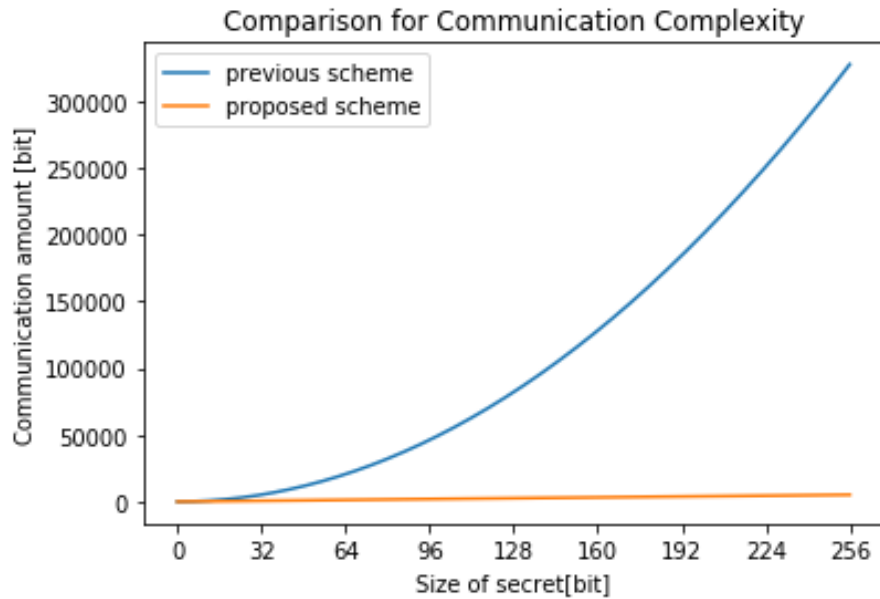


Figure 6.1: Comparison for communication bits between previous scheme and scheme 1 in this paper

6.3 Experimental Evaluation

6.3.1 Implementation Aspects

The cryptosystems which are based on discrete logarithm are basically constructed by modular exponentiations of group elements, these are suitable for our schemes. Table 6.11 shows the key sizes of discrete-log based cryptosystems which is recommended in some evaluation documents. We can see that the size of exponent is much smaller than the size of group elements. In this setting, we can apply Scheme 3 in this paper, which is most efficient one in our proposal.

Table 6.11: Appropriate data length of discrete-log based cryptosystem for 128/256-bit security

Method	discrete keys		Logarithm Group	
Security level	128	256	128	256
Lenstra/Verheul [83]	230	474	6790	49979
Lenstra Updated [82]	256	512	4440	26268
ECRYPT [48]	256	512	3072	15360
NIST [100]	256	512	3072	15360
ANSSI [101]	200	200	2048	3072
RFC3776 [102]	256	512	3253	15489

We consider a simple scenario for distributed signatures based on discrete logarithm: the key storage server is distributed three parties P_1, P_2, P_3 . Let the signing key x of the signature scheme is a element of \mathbb{Z}_q where q is prime. We assume x is shared among P_1, P_2, P_3 by the replicated secret sharing scheme. Now, a certain authorized user throw a query to the distributed server to generate own signature $\sigma \in \mathbb{Z}_p$ using shared his/her signing key by MPC, where p is prime satisfying $p > 3q + 1$ (this assumption is reasonable according to Table 6.11). As signature schemes suitable for such scenario, we can choose BLS signature [20] or Waters signature [123].

Environment and Settings We run our experiments on a cluster of three servers, each with two 10-core Intel Xeon (E5-2650 v3) processors and 128GB RAM, connected via a 10Gbps Ethernet. (We remark that little RAM was utilized and thus this is not

a parameter of importance here.)

Based on the parameter shown in Table 6.11, we run two experiments assuming 128-bit security and 256-bit security, respectively. From Table 6.10, we implement and compare Scheme 3 and previous scheme with the field of size $n = \log p$. For each experiment, we measure the latency of one MPC process of modular exponentiation, while fixing the size of q (discrete logarithm) and changing the size of p (i.e., the size of field).

We also run experiments for various network latency using `tc`¹ command on Linux. In the experiments, we tried three latency settings assuming LAN/WAN: 0.1ms, 5ms and 50ms. We suppose that 0.1ms is very low-latency of LAN, 5ms is round-trip delay of 500km distance (e.g., between Tokyo-Osaka), and 50ms is round-trip delay of 5000km distance (e.g., between Los Angeles-New York)².

6.3.2 Results and Discussion

The results on Scheme 3 are shown in Figure 6.2 and Figure 6.3. We can see that the proposed scheme works even on WAN network at the same speed as the LAN network latency. This characteristic comes from that the round complexity is constant for the size of the field. On the other hand, the previous scheme takes $O(n)$ rounds and therefore the latency is much larger than the result in Figure. 6.2 and 6.3. For example, if the size of field is 2048, the round complexity of proposed scheme is $3 \cdot 2048 + 2 = 6146$ according to Table 6.11. It takes $0.1 \cdot 6,146 = 614.6\text{ms}$ when the case of 0.1ms-latency network, and $50 \cdot 6146 = 307,300\text{ms}$ when the case of 50ms-latency network ignoring the computation cost etc. In the worst case of this experiment, which is the $\log p = 27,648$ bits with 50ms-latency network for 256-bit security, it takes $50 \cdot (3 \cdot 27,648 + 2) = 4,147,300\text{ms} = 69.1$ minutes for *only* network delay. Namely, our proposed scheme is roughly one or two order of magnitude faster than previous scheme in a certain setting.

¹This command allows us to show/change network traffic settings, like latency, packet loss, etc. (The name means “traffic control”.)

²We assume the speed of the light passing through the optical fiber is roughly 200,000km/s

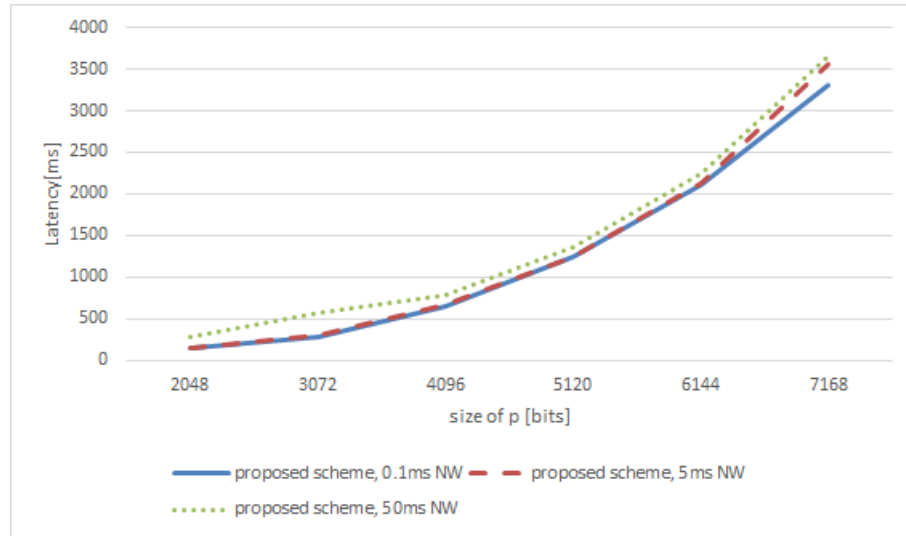


Figure 6.2: Latency-field size with 128-bit security parameter ($\log q = 256$ bit)

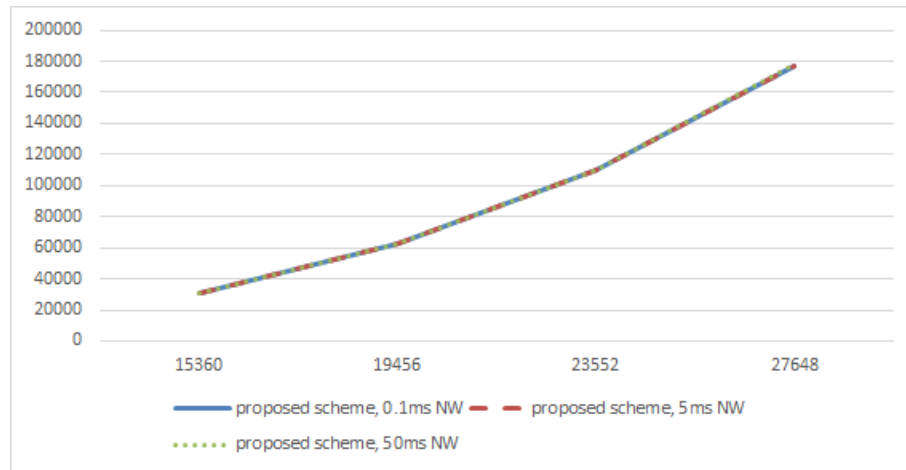


Figure 6.3: Latency-field size with 256-bit security parameter ($\log q = 512$ bit)

Chapter 7 Conclusion

Finally, we conclude this thesis.

In this thesis, we focus on the question of achieving MPC protocols with *very high throughput* on a *fast network*. This challenge in achieving this is both on the computational and network levels, and both on theory and implementation levels. In particular, we have been focused on achieving limits of high-throughput MPC and its implementation in the settings that 3-party, honest-majority and basically assuming only information-theoretic assumptions, and utilizing cache memories and CPU instructions for vectorization to reduce computation cost.

The results achieving in this thesis are the following:

High-throughput semi-honest secure 3-party computation based on replicated SS with honest-majority: We describe a new information-theoretic protocol (and a computationally-secure variant) for secure 3-party computation with an honest majority. The protocol has very minimal computation and communication; for Boolean circuits, each party sends only a single bit for every AND gate (and nothing is sent for XOR gates).

Optimizing cheating detection for honest-majority MPC We improve general techniques for cheater detection protocol in MPC, which is based on cut-and-choose protocols on multiplication triples. In addition, we utilize them to significantly improve the recently published protocol of Furukawa et al. We reduce the bandwidth of their protocol down from 10 bits per AND gate to 7 bits per AND gate, and show how to improve some computationally expensive parts of their protocol. Our implementation achieves a rate of approximately 1.15 billion AND gates per second on a cluster of three 20-core machines with a 10Gbps network. Thus, we can securely compute 212,000 AES encryptions per second (which is hundreds of times faster than previous work for this setting). Our results demonstrate that high-throughput secure computation for malicious adversaries is possible.

Compiler for secret-sharing based secure computation We design and implement a MPC compiler for our three-party honest majority MPC. Our implementation

is an extension of a well-known MPC compiler called “SPDZ compiler” so that it can work with general underlying protocols. In this thesis we called the compiler we made “generalized SPDZ compiler”. Moreover, our SPDZ extensions were made in mind to enable the use of SPDZ for arbitrary protocols and to make it easy for others to integrate existing and new protocols.

Dedicated MPC protocol for complex functionalities : the cases of bit decomposition, ring composition and modular exponentiation We propose RSSS-based three party computation protocols for (1) bit decomposition; namely, arithmetic-to-Boolean type conversion (2) ring composition; namely, Boolean-to-arithmetic type conversion (3) modular exponentiation on the case where the base is public and the exponent is private.

Compared with the previous best protocols, our bit decomposition and ring composition achieve two order of magnitude less communication bits in 32-bit integer case, which is considered as a reasonable parameter. The protocols are integrated into the generalized SPDZ compiler described above and thus we can see the practical efficiency of these protocols in the complex mixed operation of arithmetic and Boolean, like SQL query on fixed-point numbers.

Regarding modular exponentiation, we will show the practical effect of our protocol by experiments on the scenario for distributed signatures, which is useful for secure key management on the distributed environment (e.g., distributed ledgers). Our modular exponentiation protocols are more efficient in terms of both of communication complexity and round complexity than previous standard scheme. More precisely, for the size of secret values n , the proposed schemes require $O(n)$ bits communication whereas the previous scheme requires $O(n^2)$ bits. As for the round complexity, a several variants in our proposal require $O(n)$ round as same as previous scheme, and other variants in our proposal require just $O(1)$ rounds.

References

- [1] Intel Haswell cache performance. <http://www.7-cpu.com/cpu/Haswell.html>.
- [2] Rakesh Agrawal, Alexandre V. Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 86–97, 2003.
- [3] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 439–450, 2000.
- [4] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure Computation on Floating Point Numbers. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.
- [5] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 843–862, 2017.
- [6] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 805–817, 2016.

- [7] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Kelle, Yehuda Lindell, Ariel Nof, Kazuma Ohara and Hikaru Tsuchida. Generalizing the SPDZ Compiler For Other Protocols. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 880–895, 2018.
- [8] Gilad Asharov and Yehuda Lindell. A Full Proof of the BGW Protocol for Perfectly-Secure Multiparty Computation. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:36, 2011.
- [9] Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Osama Khan, Lea Kissner, Zachary N. J. Peterson, and Dawn Song. Remote data checking using provable data possession. *ACM Trans. Inf. Syst. Secur.*, 14(1):12:1–12:34, 2011.
- [10] Donald Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, pages 420–432, 1991.
- [11] Donald Beaver, Joan Feigenbaum, Joe Kilian, and Phillip Rogaway. Security with low communication overhead. In *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, pages 62–76, 1990.
- [12] Donald Beaver and Shafi Goldwasser. Multiparty Computation with Faulty Majority. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 589–590, 1989.
- [13] Donald Beaver, Silvio Micali, and Phillip Rogaway. The Round Complexity of Secure Protocols (Extended Abstract). In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 503–513, 1990.

- [14] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.
- [15] Josh Cohen Benaloh and Dwight Tuinstra. Receipt-free secret-ballot elections (extended abstract). In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*, pages 544–553, 1994.
- [16] G. R. Blakley. Safeguarding cryptographic keys. In *Managing Requirements Knowledge, International Workshop on (AFIPS)*, volume 00, page 313, 12 1899.
- [17] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, pages 192–206, 2008.
- [18] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.
- [19] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure Multiparty Computation Goes Live. In *Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers*, pages 325–343, 2009.
- [20] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptology*, 17(4):297–319, 2004.
- [21] C. Boyd. Digital Multisignatures. *Cryptography and Coding*.
- [22] Colin Boyd. A new multiple key cipher and an improved voting scheme. In *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and*

- Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings*, pages 617–625, 1989.
- [23] Felix Brandt. How to obtain full privacy in auctions. *Int. J. Inf. Sec.*, 5(4):201–216, 2006.
 - [24] Bristol Cryptography Group. SPDZ software. <https://www.cs.bris.ac.uk/Research/CryptographySecurity/SPDZ/>, 2016.
 - [25] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High Performance Multi-Party Computation for Binary Circuits Based on Oblivious Transfer. *IACR Cryptology ePrint Archive*, 2015:472, 2015.
 - [26] Niklas Büscher, Andreas Holzer, Alina Weber, and Stefan Katzenbeisser. Compiling Low Depth Circuits for Practical Secure Computation. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part II*, pages 80–98, 2016.
 - [27] Niklas Büscher and Stefan Katzenbeisser. *Compilation for Secure Multi-party Computation*. Springer Briefs in Computer Science. Springer, 2017.
 - [28] Ran Canetti. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptology*, 13(1):143–202, 2000.
 - [29] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145, 2001.
 - [30] Octavian Catrina and Sebastiaan de Hoogh. Secure Multiparty Linear Programming Using Fixed-Point Arithmetic. In *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings*, pages 134–150, 2010.
 - [31] Octavian Catrina and Amitabh Saxena. Secure Computation with Fixed-Point Numbers. In *Financial Cryptography and Data Security, 14th International*

- Conference, FC 2010, Tenerife, Canary Islands, Spain, January 25-28, 2010, Revised Selected Papers*, pages 35–50, 2010.
- [32] David Chaum. Untraceable electronic mail, return addresses and digital pseudonyms. In *Secure Electronic Voting*, pages 211–219. 2003.
- [33] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty Unconditionally Secure Protocols (Extended Abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 11–19, 1988.
- [34] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. In *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, pages 34–64, 2018.
- [35] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults (Extended Abstract). In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 383–395, 1985.
- [36] Thomas M. Cover. Broadcast channels. *IEEE Trans. Information Theory*, 18(1):2–14, 1972.
- [37] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share Conversion, Pseudo-random Secret-Sharing and Applications to Secure Computation. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, pages 342–362, 2005.
- [38] R.A. Croft and S.P. Harris. Public-Key Cryptography and Reusable Shared Secrets. *Cryptography and Coding*.
- [39] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. In *Theory of Cryptography*,

- Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, pages 285–304, 2006.
- [40] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous Multiparty Computation: Theory and Implementation. In *Public Key Cryptography - PKC 2009, 12th International Conference on Practice and Theory in Public Key Cryptography, Irvine, CA, USA, March 18-20, 2009. Proceedings*, pages 160–179, 2009.
- [41] Ivan Damgård and Marcel Keller. Secure Multiparty AES. In *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, Spain, January 25-28, 2010, Revised Selected Papers*, pages 367–374, 2010.
- [42] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, pages 1–18, 2013.
- [43] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 643–662, 2012.
- [44] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.
- [45] Yvo Desmedt. Society and group oriented cryptography: A new concept. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, pages 120–127, 1987.
- [46] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference,*

- Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, pages 307–315, 1989.
- [47] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Information Theory*, 22(6):644–654, 1976.
- [48] ECRYPT-CSA. Algorithms, key size and protocols report(2018). H2020-ICT-2014 – Project 645421, 2018.
- [49] Herbert Enderton. A mathematical introduction to logic (2nd ed.). Boston, MA: Academic Press, ISBN 978-0-12-238452-3., 2001.
- [50] Shimon Even, Oded Goldreich, and Abraham Lempel. A Randomized Protocol for Signing Contracts. *Commun. ACM*, 28(6):637–647, 1985.
- [51] Ronald Fisher and Frank Yates. Statistical Tables for Biological, Agricultural and Medical Research (3rd ed.). Oliver & Boyd., 1938.
- [52] Matthew K. Franklin and Michael K. Reiter. The design and implementation of a secure auction service. *IEEE Trans. Software Eng.*, 22(5):302–312, 1996.
- [53] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations. In *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 244–249, 2014.
- [54] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 1–19, 2004.
- [55] Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. A practical secret voting scheme for large scale elections. In *Advances in Cryptology - AUSCRYPT '92, Workshop on the Theory and Application of Cryptographic Techniques, Gold Coast, Queensland, Australia, December 13-16, 1992, Proceedings*, pages 244–251, 1992.

- [56] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*, pages 225–255, 2017.
- [57] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-Optimal DSA/ECDSA Signatures and an Application to Bitcoin Wallet Security. In *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*, pages 156–174, 2016.
- [58] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Robust threshold DSS signatures. *Inf. Comput.*, 164(1):54–84, 2001.
- [59] Oded Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.
- [60] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987.
- [61] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The Knowledge Complexity of Interactive Proof Systems. *SIAM J. Comput.*, 18(1):186–208, 1989.
- [62] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. Fast Garbling of Circuits Under Standard Assumptions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 567–578, 2015.
- [63] Michael Harkavy, J. Doug Tygar, and Hiroaki Kikuchi. Electronic auctions with private bids. In *Proceedings of the 3rd USENIX Workshop on Electronic Commerce, Boston, Massachusetts, USA, August 31 - September 3, 1998*, 1998.

- [64] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low Cost Constant Round MPC Combining BMR and Oblivious Transfer. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, pages 598–628, 2017.
- [65] McKinsey Global Institute. The ‘big data’ revolution in US healthcare: accelerating value and innovation, January, 2013.
- [66] McKinsey Global Institute. Big data: The next frontier for innovation, competition, and productivity, May, 2011.
- [67] Yuval Ishai and Eyal Kushilevitz. On the Hardness of Information-Theoretic Multiparty Computation. In *Advances in Cryptology - EUROCRYPT 2004, International Conference on the Theory and Applications of Cryptographic Techniques, Interlaken, Switzerland, May 2-6, 2004, Proceedings*, pages 439–455, 2004.
- [68] Marcel Keller. The Oblivious Machine - or: How to Put the C into MPC. *IACR Cryptology ePrint Archive*, 2015:467, 2015.
- [69] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively Secure OT Extension with Optimal Overhead. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, pages 724–741, 2015.
- [70] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 830–842, 2016.
- [71] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ Great Again. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, pages 158–189, 2018.

- [72] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 549–560, 2013.
- [73] Marcel Keller and Avishay Yanai. Efficient Maliciously Secure Multiparty Computation for RAM. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, pages 91–124, 2018.
- [74] Marcel Keller and Avishay Yanay. ORAM in SPDZ-BMR. <https://github.com/mkskeller/SPDZ-BMR-ORAM>, 2018.
- [75] Liisi Kerik, Peeter Laud, and Jaak Randmets. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In *Financial Cryptography and Data Security - FC 2016 International Workshops, 4th WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers*, pages 271–287, 2016.
- [76] Florian Kerschbaum. Outsourced private set intersection using homomorphic encryption. In *7th ACM Symposium on Information, Computer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012*, pages 85–86, 2012.
- [77] Ryo Kikuchi, Dai Ikarashi, Takahiro Matsuda, Koki Hamada, and Koji Chida. Efficient Bit-Decomposition and Modulus-Conversion Protocols with an Honest Majority. In *Information Security and Privacy - 23rd Australasian Conference, ACISP 2018, Wollongong, NSW, Australia, July 11-13, 2018, Proceedings*, pages 64–82, 2018.
- [78] Eyal Kushilevitz, Yehuda Lindell, and Tal Rabin. Information-Theoretically Secure Protocols and Security under Composition. *SIAM J. Comput.*, 39(5):2090–2112, 2010.
- [79] Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest Majority Multi-Party Computation for Binary Circuits. In *Advances in Cryptology*

- *CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 495–512, 2014.
- [80] John Launchbury, Iavor S. Diatchki, Thomas DuBuisson, and Andy Adams-Moran. Efficient lookup-table protocol in secure multiparty computation. In *ACM SIGPLAN International Conference on Functional Programming, ICFP’12, Copenhagen, Denmark, September 9-15, 2012*, pages 189–200, 2012.
- [81] Sven Laur, Riivo Talviste, and Jan Willemson. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings*, pages 84–101, 2013.
- [82] Arjen K. Lenstra. Key lengths. *The handbook of Information Security*, 2004.
- [83] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. *Journal Of Cryptology*, vol.14, p.255-293, 2001.
- [84] Yehuda Lindell. Fast Secure Two-Party ECDSA Signing. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part II*, pages 613–644, 2017.
- [85] Yehuda Lindell and Ariel Nof. A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 259–276, 2017.
- [86] Yehuda Lindell and Ariel Nof. Fast secure multiparty ECDSA with practical distributed key generation and applications to cryptocurrency custody. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1837–1854, 2018.
- [87] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. In *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology*

- Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, pages 36–54, 2000.
- [88] Yehuda Lindell, Benny Pinkas, Nigel P. Smart, and Avishay Yanai. Efficient Constant Round Multi-party Computation Combining BMR and SPDZ. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II*, pages 319–338, 2015.
- [89] Helger Lipmaa, N. Asokan, and Valtteri Niemi. Secure vickrey auctions without threshold trust. In *Financial Cryptography, 6th International Conference, FC 2002, Southampton, Bermuda, March 11-14, 2002, Revised Papers*, pages 87–101, 2002.
- [90] Philip D. MacKenzie and Michael K. Reiter. Two-party generation of DSA signatures. *Int. J. Inf. Sec.*, 2(3-4):218–239, 2004.
- [91] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - Secure Two-Party Computation System. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 287–302, 2004.
- [92] Payman Mohassel and Peter Rindal. ABY3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 35–52, 2018.
- [93] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 591–602, 2015.
- [94] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 1985.

- [95] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the First ACM Conference on Electronic Commerce (EC-99), Denver, CO, USA, November 3-5, 1999*, pages 129–139, 1999.
- [96] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security, Philadelphia, Pennsylvania, USA, November 6-8, 2001.*, pages 116–125, 2001.
- [97] Chao Ning and Qiuliang Xu. Multiparty computation for modulo reduction without bit-decomposition and a generalization to bit-decomposition. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*, pages 483–500, 2010.
- [98] Chao Ning and Qiuliang Xu. Constant-rounds, linear multi-party computation for exponentiation and modulo reduction with perfect security. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, pages 572–589, 2011.
- [99] Takashi Nishide and Kazuo Ohta. Multiparty Computation for Interval, Equality, and Comparison Without Bit-Decomposition Protocol. In *Public Key Cryptography - PKC 2007, 10th International Conference on Practice and Theory in Public-Key Cryptography, Beijing, China, April 16-20, 2007, Proceedings*, pages 343–360, 2007.
- [100] NIST. Recommendation for key management. Special Publication 800-57 Part 1 Rev.4, 2016.
- [101] NSA. Mécanismes cryptographiques. Règles et recommandations, Rev. 2.03, 2014.
- [102] NSA. Commercial national security algorithms. Information Assurance Directorate at the NSA, 2016.

- [103] National Bureau of Standards. Data Encryption Standard, FIPS-Pub.46. National Bureau of Standards. U.S. Department of Commerce, Washington D.C., January 1977.
- [104] Kazuma Ohara, Yohei Watanabe, Mitsugu Iwamoto, Kazuo Ohta. Multi-Party Computation for Modular Exponentiation based on Replicated Secret Sharing. In *IEICE Transaction, Vol.E102-A, No.9, Sep. 2019*. (to appear)
- [105] Tatsuaki Okamoto. Receipt-free electronic voting schemes for large scale elections. In *Security Protocols, 5th International Workshop, Paris, France, April 7-9, 1997, Proceedings*, pages 25–35, 1997.
- [106] Haijun Pan, Edwin S. H. Hou, and Nirwan Ansari. Enhanced name and vote separated e-voting system: an e-voting system that ensures voter confidentiality and candidate privacy. *Security and Communication Networks*, 7(12):2335–2344, 2014.
- [107] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: a centralized "zero-queue" datacenter network. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, pages 307–318, 2014.
- [108] Michael O. Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive*, 2005:187, 2005.
- [109] Tal Rabin and Michael Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority (Extended Abstract). In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washington, USA*, pages 73–85, 1989.
- [110] Jaak Randmets. AES performance on the new Sharemind cluster. Personal comm. May, 2016.
- [111] Peter Rindal and Mike Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 297–314, 2016.

- [112] Adi Shamir. How to Share a Secret. *Commun. ACM*, 22(11):612–613, 1979.
- [113] Adi Shamir, Ronald L. Rivest, and Leonard M. Adleman. Mental Poker. The Mathematical Gardner, Belmont, Cali., Wadsworth International.
- [114] Victor Shoup. Practical threshold signatures. In *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, pages 207–220, 2000.
- [115] Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *J. Cryptology*, 15(2):75–96, 2002.
- [116] Vivek Kumar Singh, Burcin Bozkaya, and Alex Pentland. Money Walks: Implicit Mobility Behavior and Financial Well-Being, PLoS ONE 10(8): e0136628. <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0136628>, 2015.
- [117] Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 411–428, 2015.
- [118] Riivo Talviste. Applying Secure Multi-Party Computation in Practice. Ph.D dissertation, University of Tartu, 2016.
- [119] Tomas Toft. Constant-Rounds, Almost-Linear Bit-Decomposition of Secret Shared Values. In *Topics in Cryptology - CT-RSA 2009, The Cryptographers’ Track at the RSA Conference 2009, San Francisco, CA, USA, April 20-24, 2009. Proceedings*, pages 357–371, 2009.
- [120] Jaideep Vaidya and Chris Clifton. Secure set intersection cardinality with application to association rule mining. *Journal of Computer Security*, 13(4):593–622, 2005.
- [121] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation. In *Proceedings of the*

- 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 21–37, 2017.
- [122] Yujue Wang, Duncan S. Wong, Qianhong Wu, Sherman S. M. Chow, Bo Qin, and Jianwei Liu. Practical Distributed Signatures in the Standard Model. In *Topics in Cryptology - CT-RSA 2014 - The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings*, pages 307–326, 2014.
- [123] Brent Waters. Efficient Identity-Based Encryption Without Random Oracles. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, pages 114–127, 2005.
- [124] Wikipedia. Carry-Select Adder. https://en.wikipedia.org/wiki/Carry-select_adder, March 2015.
- [125] Andrew Chi-Chih Yao. Protocols for Secure Computations (Extended Abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164, 1982.
- [126] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets (Extended Abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167, 1986.
- [127] Samee Zahur, Mike Rosulek, and David Evans. Two Halves Make a Whole - Reducing Data Transfer in Garbled Circuits Using Half Gates. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*, pages 220–250, 2015.

List of Publications

Related Papers

Journal Papers

1. Kazuma Ohara, Yohei Watanabe, Mitsugu Iwamoto, Kazuo Ohta: “Multi-Party Computation for Modular Exponentiation based on Replicated Secret Sharing.” IEICE Transaction, Vol.E102-A,No.9,Sep. 2019. (to appear)

Refereed Conference Papers (with Formal Proceedings)

2. Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, Kazuma Ohara: “High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority.” ACM Conference on Computer and Communications Security 2016: 805-817.
3. Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, Or Weinstein: “Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier. ” IEEE Symposium on Security and Privacy 2017: 843-862.
4. Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, Hikaru Tsuchida: “Generalizing the SPDZ Compiler For Other Protocols. ” ACM Conference on Computer and Communications Security 2018: 880-895.

Referred Papers

Refereed Conference Papers, Posters and Demo (with Formal Proceedings)

5. Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Kazuma Ohara, Hikaru Tsuchida: “How to Choose Suitable Secure Multiparty Computation Using Generalized SPDZ.” ACM Conference on Computer and Communications Security 2018 (ACM CCS 2018): 2198-2200.
6. Toshinori Araki, Assaf Barak, Jun Furukawa, Yehuda Lindell, Ariel Nof, Kazuma Ohara: DEMO: “High-Throughput Secure Three-Party Computation of Kerberos Ticket Generation.” ACM Conference on Computer and Communications Security 2016: 1841-1843.

Non-Refereed Papers

7. 大原一真, 荒木敏則, 土田光, 古川潤: “異なるサイズの環が混在する不正検知可能なマルチパーティ計算”, 暗号と情報セキュリティシンポジウム 2018(SCIS2018), 2A1-4, 2018 年.
8. 土田光, 荒木敏則, 大原一真, 古川潤: ”不正検知可能なマルチパーティー計算による生体情報と遺伝子情報の保護”, 暗号と情報セキュリティシンポジウム 2018(SCIS2018), 2A1-5, 2018 年.
9. 荒木俊則, 古川潤, Yehuda Lindell, Ariel Nof, 大原一真. ”通信量の小さい 3 者間マルチパーティ計算とビットスライス法による実装”, 暗号と情報セキュリティシンポジウム 2017 (SCIS2017), 2D3-5, 2017 年.
10. Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, Hikaru Tsuchida: “Generalizing the SPDZ Compiler For Other Protocols.” IACR Cryptology ePrint Archive 2018: 762 (2018). Available from <https://eprint.iacr.org/2018/762>.
11. Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, Kazuma Ohara: “High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority.” IACR Cryptology ePrint Archive 2016: 768 (2016). Available from <https://eprint.iacr.org/2016/768>.

Review Articles

12. 大原一真: “秘密分散法を用いた秘密計算” システム制御情報学会誌「システム/制御/情報」プライバシー保護データマイニング特集号, Vol.63, No.2, 2019.

Author Biography

Kazuma Ohara was born in Kanagawa, Japan, on October 9, 1989. He received his B.E. degree from the University of Electro-Communications, Tokyo, Japan, in 2012, and his M.E. degree from the University of Electro-Communications, Tokyo, Japan, in 2014, respectively. Since 2014, he has been working for NEC Central Research Laboratories, NEC Corporation. Since 2017, he has been a Ph.D course student at the Graduate School of Informatics and Engineering, the University of Electro-Communications. He is engaged in research on cryptography. His research interests include not only secure multi-party computation but design, implementation and proof of security on cryptographic protocols such as group signatures, searchable encryptions. He was awarded a paper prize from the 2014 Symposium on Cryptography and Information Security (SCIS 2014) in 2015, and a best paper award from the 23rd ACM Conference on Computer and Communications Security (ACM CCS 2016) in 2016.