UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Multi-Party Computation as a Service for Privacy-Preserving Distributed Applications

Miguel João Novo Faísca de Carvalho

Mestrado em Engenharia Informática

Dissertação orientada por:
Prof. Doutor Bernardo Luís da Silva Ferreira
Prof. Doutor Alysson Neves Bessani

2024

Acknowledgements

I would like to thank my advisors, Bernardo and Alysson, for their invaluable guidance and support throughout the development of this thesis. Their feedback greatly contributed to the quality of this work, and I am also immensely thankful for their patience, mentorship, and belief in my abilities.

I would also like to express my gratitude to Robin Vassantlal for his constructive comments and assistance. His help was not only greatly appreciated but also crucial, as it guided me in learning about complex topics related to this work and managing the COBRA library more efficiently. I deeply thank the time and effort he devoted to aiding me to overcome challenges and broaden my understanding of key topics.

Finally, the development of this work was made possible with the support of FCT through the SMaRtChain project, ref. 2022.08431.PTDC (https://doi.org/10.54499/2022.08431.PTDC), the LASIGE Research Unit, ref. UIDB/00408/2020 (https://doi.org/10.54499/UIDB/00408/2020) and ref. UIDP/00408/2020 (https://doi.org/10.54499/UIDP/00408/2020).

To the Atlas and Hestia of my life

Resumo

A Computação Segura entre Pares (CSeP) permite que vários utilizadores pertencentes a uma rede distribuída consigam realizar operações sobre dados cifrados de forma a preservar a privacidade e confidencialidade destes. Uma forma de implementar CSeP é através de Partilha de Segredos (PS), uma técnica criptográfica que permite dividir segredos de utilizadores em várias partes (chamadas de *shares*) de tal maneira que o segredo pode ser reconstruído quando um número predefinido de shares é combinado. Vários Esquemas de Partilha de Segredos, como o de Shamir, têm a vantagem de ser possível realizar operações diretamente sobre shares de segredos como se estivesse a operar sobre os segredos originais, possibilitando operações aritméticas como adições e multiplicações. Assumindo este contexto, implementar um CSeP seguro e eficiente para cenários práticos impõe alta complexidade. Tal deve-se ao facto de o CSeP tradicional assumir um conjunto fixo de participantes, no qual os utilizadores são obrigados a suportar todo o esforço computacional independentemente dos recursos que lhes estejam disponíveis. Consequentemente, estes CSeP's não toleram computações de larga escala, dado o intervalo de tempo substancial associado à computação que impõe restrições aos participantes. Para lidar com estes problemas, estudos recentes adotaram uma abordagem dinâmica, na qual os participantes de CSeP podem se juntar ou sair da computação sem interromper a execução do protocolo. Para grandes computações, mecanismos proativos permitem a renovação/recuperação de shares, assegurando a segurança do sistema contra adversários *móveis*. Além disto, outros trabalhos exploram o modelo de CSeP-como-serviço que define clientes (que iniciam os pedidos) e servidores (que realizam as operações solicitadas), algo que aumenta flexibilidade ao separar as tarefas de partilha de segredos e computação. Mesmo assim, os protocolos atuais tendem a seguir um modelo dinâmico baseado em comunicação entre comités de participantes de diferentes rondas computacionais, algo que não se adapta bem em ambientes assíncronos devido a possíveis atrasos na rede.

Apresentamos MPCServe, uma framework de CSeP dinâmico que segue o modelo CSeP-como-serviço para permitir computações práticas em ambientes assíncronos. O nosso protocolo propõe clientes que solicitam operações, representadas em circuitos aritméticos, e um conjunto de servidores que computam as operações solicitadas pelos clientes. O MPCServe corre sobre o COBRA, uma framework confidencial de Replicação de Máquinas de Estado Tolerante a Faltas Bizantinas (RME-TFB), que utiliza Partilha de Segredos Proativa e Dinâmica (PSPD) baseada no esquema de Shamir para garantir a confidencialidade das shares mantidas pela Key-Value (KV) store de cada servidor. Desta forma, definimos um sistema de CSeP seguro contra adversários

totalmente maliciosos e que apresenta propriedades de tolerância a falhas, substituição de participantes flexível, e garantia de outputs sob a rede assíncrona do COBRA.

O MPCServe assume um ambiente distribuído composto por um conjunto de processos divididos em n servidores e um conjunto de clientes que solicitam operações ao conjunto de servidores. O sistema pressupõe uma configuração inicial e segura com identificadores públicos únicos e um modelo parcialmente síncrono. A comunicação ocorre através de canais privados, autenticados e justos. O *modelo adversarial* da nossa solução considera um adversário adaptativo em tempo polinomial probabilístico, que é capaz de corromper até $t < n/3$ das réplicas servidores. Da mesma forma que o protocolo COBRA, consideramos que os clientes que acedem ao sistema associado ao MPCServe são honestos (seguem a estrutura correta do protocolo). O *modelo de serviço* do MPCServe é definido por: uma fase de inicialização, onde os clientes solicitam operações ao conjunto de servidores que participam no CSeP; uma fase de computação, onde os servidores tratam de pedidos dos clientes por meio do armazenamento ou obtenção de dados contidos na Key-Value store distribuída ou computando uma operação representada como um circuito aritmético; e uma fase de conclusão, onde os servidores devolvem os resultados da operação computada em forma de shares aos clientes envolvidos.

Os circuitos enviados por clientes são avaliados pelos servidores do MPCServe porta a porta. As portas de adição e adição/multiplicação por constantes são avaliadas localmente, nas quais os servidores operam diretamente sobre as suas shares (dada a propriedade de adição homomórfica do esquema de Shamir) e criam shares resultantes dessa operação sem comunicação adicional. No entanto, as portas de multiplicação introduzem um grau de complexidade adicional, já que a multiplicação de shares correspondentes a polinómios de grau t leva a shares de output pertencentes a polinómios de maior grau, algo que quebra o limite t pré-definido no esquema de partilha de segredos. De forma a resolver isto, o sistema segue uma versão otimizada de um protocolo tradicional de CSeP, de onde os servidores multiplicam localmente as suas shares para obter uma share de multiplicação de maior grau, criam e partilham sub-shares de polinómios baseados na share multiplicada, e realizam uma equação linear sobre as sub-shares recebidas para obter shares de multiplicação de grau correto. Mesmo assim, o protocolo de multiplicações requer alterações de forma a torná-lo viável para ambientes práticos. Isto deve-se ao facto de a solução assumir que $2t + 1$ servidores participam no processo de criação de sub-shares, embora possam existir mais dada a discrepância com o modelo que existe com o modelo RME-TFB que assume $3t + 1$ replicas. Também, este protocolo de multiplicação requer todas as sub-shares criadas por estes $2t + 1$ servidores de forma que seja possível gerar uma share de multiplicação por se trabalhar sobre polinómios criados por diferentes servidores (ao invés da partilha de segredos tradicional que assume apenas um polinomio). Desta feita, o MPCServe introduz o conceito de *justiça* para permitir rodar as responsabilidades de execução entre servidores e evitar que os mesmos $2t + 1$ (em cenários onde $n > 2t + 1$) lidem com toda a rotina de multiplicação. Para também garantir a *recuperação eficiente de shares*, são utilizadas sub-rotinas baseadas em *propostas*, que encapsulam sub-shares através de encriptação assimétrica, permitindo assim a partilha segura destas entre

os servidores. Como é impossível distinguir entre servidores honestos que se atrazam e servidores maliciosos em ambientes assíncronos, a *tolerância a faltas* é gerida através de um algoritmo ($Mult_{Delay}$) que lida com atrasos nas mensagens e crash de servidores. Para garantir segurança contra adversários maliciosos, o MPCServe recorre ao uso de Feldman commitments e propõe um algoritmo ($Mult_{Accuse}$) para resolver shares inválidas enviadas por servidores maliciosos. Por fim, apresentamos uma prova de Non-Interactive Zero Knowledge (NIZK) para permitir que servidores consigam provar de forma efetiva a posse de shares corretas durante o processo de multiplicação.

Os testes feitos ao MPCServe utilizam configurações realistas assumindo 4,7, e 10 máquinas distribuídas. Em concreto, um conjunto de máquinas Dell PowerEdge R410, com 32GB de memória e processadores 2-QuadCore Intel Xeon E5520 de 2,27 GHz foram utilizadas. Resultados mostram que o desempenho médio de uma porta de adição (assumindo 1000 iterações) mantém-se a uma velocidade rápida e consistente de $1ms$, enquanto as portas de multiplicação reportam uma latência superior — $70ms$ para $t = 1$, $116ms$ para $t = 2$ e $185ms$ para $t = 3$. Um benchmark de vários circuitos corrobora o previamente mencionado: a relação observada é de $0.168/70$, o que significa que as adições são aproximadamente 0.0024 vezes mais rápidas do que as multiplicações. Este desfasamento reflete-se no impacto que as multiplicações têm sobre o tempo total de execução de diferentes circuitos, já que circuitos compostos maioritariamente por adições (como a média) são executados de forma extremamente rápida dado a grande escala de operações locais, enquanto que circuitos envolvendo várias multiplicações (como o produto interno) tornam-se exponencialmente mais lentos devido à comunicação entre servidores, verificação de sub-shares partilhadas, e outros mecanismos associados ao processo de uma multiplicação. Um teste do produto interno de dois vetores demonstra também que o sistema apresenta um bom desempenho — $72, 118, 196$ segundos para $t = 1, 2, 3$ — para dimensões de vetor menores ($<= 1000$), alcançando uma velocidade semelhante ($\sim 1.006\%$) às portas de multiplicação individuais. Para maiores dimensões, a latência aumenta aproximadamente 69% de $t = 1$ para $t = 2$ e cerca de 92% de $t = 2$ para $t = 3$, devido ao aumento de operações de partilha de segredos, troca de mensagens e verificação de provas. Finalmente, a performance das multiplicações perante um crash ou um adversário malicioso que partilha sub-shares inválidas apresenta um aumento de 97% comparativamente ao cenário normal dado à necessidade do sistema gerir o servidor que falhou e refazer a multiplicação assumindo um novo grupo de servidores.

Palavras-chave: Computação Segura entre Pares, Partilha de Segredos, Tolerância a Faltas Bizantinas

Abstract

Multi-Party Computation (MPC) has recently gained interest as a tool to perform secure, distributed computations. However, current work presents limitations that hinder their practical application, namely not supporting a framework suited for long computations, assuming a fixed participant size, not tolerating faults, and running in strictly synchronous environments. We present MPCServe, a new practical Multi-Party Computation framework that dynamically performs computations while adopting a *MPC-as-a-Service* model for ease of usage. Our framework allows lightweight clients to outsource their privacy-preserving computations on encrypted data to a set of untrusted servers while guaranteeing computational output in the presence of t Byzantine faults assuming a total of at least $n > 3t$ servers. MPCServe extends *COBRA*, a confidential Byzantine Fault Tolerance State Machine Replication framework that uses Dynamic Proactive Secret Sharing (DPSS) for storing data with high levels of privacy, integrity, and availability. Leveraging its fault-tolerance guarantees and the homomorphic properties of DPSS, MPCServe builds a fluid-style, maliciously secure MPC infrastructure for asynchronous networks that allows servers to join and leave during the computational effort.

Keywords: Secret Sharing, Secure Multi-party Computation, Byzantine Fault Tolerance

Contents

List of Figures

List of Tables

Acronyms

ACS Agreement on a Core Set.

BA Byzantine Agreement.

BFT SMR Byzantine Fault Tolerant State Machine Replication.

DPSS Dynamic Proactive Secret Sharing.

FHE Fully Homomorphic Encryption.

KV Key-Value.

MAC Message Authentication Code.

MPC Multi-Party Computation.

NIZK Non Interactive Zero Knowledge.

OT Oblivious Transfer.

PPT Probabilistic Polynomial-Time.

PSS Proactive Secret Sharing.

SMR State Machine Replication.

SS Secret Sharing.

VSS Verifiable Secret Sharing.

ZK Zero Knowledge.

Chapter 1

Introduction

1.1 Context and Motivation

In today's age, the exponential growth of digital information has reached unprecedented levels, with an estimated 147 zettabytes expected to be generated by the end of 2024. This surge in data production, fueled by its rapid consumption of big data, cloud computing, and the Internet of Things (IoT), has made the need for privacy-preserving data processing methods increasingly urgent.

To address this, cryptography (i.e. key-based encryption) plays a central role in ensuring the protection of data against cybersecurity threats, security breaches, and other unauthorized access. While effective, these mechanisms either render operations over encrypted data impossible or pose significant complexity to the process, hindering the work of organizations that rely on resource sharing to operate effectively.

Consequently, Multi-Party Computation (MPC) [82] protocols have become increasingly essential in addressing the problem of cooperative computation, as they allow untrusted entities to work together in a distributed setting and perform computations over private data without compromising its confidentiality. MPC provides the most versatile approach to secure distributed computation, seeing interest in areas such as distributed key generation [66], statistical analysis [36], blockchains [50], and machine learning purposes [53].

Still, there exists a strong motivation to make MPC more practical and deployable in realistic scenarios where asynchronous environments and fault tolerance are presumed, a push supported by the current trend for the global MPC market size, which is expected to grow 73.46% from 2024 to 2029 (814 to 1412 million USD)[1].

1.2 Problem

Despite the current capabilities of multi-party computation protocols, designing and implementing secure protocols for practical domains presents unique challenges.

[1] https://www.marketsandmarkets.com/Market-Reports/secure-multiparty-computation-market-67797344.html

A primary issue comes from how the traditional MPC system model is defined, which considers an infrastructure with a fixed set of users that must participate in the entire computational process. Therefore, when a computation becomes significantly large or complex, its duration may pose serious issues as the underlying MPC participants may want to leave the protocol execution due to lack of resources, time constraints, or personal commitments. This restriction has limited the application of MPC in environments where flexibility and accessibility are crucial for ensuring the correct execution of the system. To address this, recent MPC protocols started adhering to a *dynamic* approach [24, 47, 72] for easier participant engagement and adaptability. In this setting, participants join and leave the computation effort without interrupting the protocol, allowing parties to volunteer their resources - which may be limited - to a large-scale computation without being committed to the entire process. Moreover, the peer-to-peer MPC network creates variability in computational efficiency by making the protocol's performance heavily dependent on the slowest, least capable participant. To resolve resource discrepancies, following a client-server model [81, 61, 3] helps decouple the tasks of user data sharing and distributed computation by empowering a server-aided setting where a series of servers work together to process client computational requests. Merging both approaches allows for MPC-as-a-service where lightweight clients request operations to a set of servers that can join, aid the computational procedure, and leave.

Even so, current dynamic frameworks still often provide weak termination guarantees (such as aborting the computation when deviations in the protocol execution occur), do not tolerate faults, and strictly run under synchronous communication, making them ill-suited for real-world usage where malicious adversaries and asynchronous networks are common.

1.3 Objectives and Solution

In light of the previous limitations, we aim to solve the following question: ≪*Can we improve current multi-party computational protocols to provide a service for practical, dynamic, and secure computations over asynchronous networks while ensuring guaranteed output against Byzantine adversaries?*≫.

As such, this thesis introduces **MPCServe**, a *Dynamic* Multi-party Computation framework that follows an *MPC-as-a-Service* architecture for practical and secure computations. More specifically, our protocol defines clients that request operations (in the form of arithmetic circuits) and a cluster of servers that collaboratively compute given client requests. MPCServe runs over *COBRA* [79], a Confidential Byzantine-Fault Tolerant State Machine Replication (BFT-SMR) [14] framework which uses Dynamic Proactive Secret Sharing (DPSS) [75] to ensure data confidentiality. Additionally, MPCServe extends Gennaro et al. [46] MPC, an optimized version of the classic BGW [11] protocol based on Shamir's Secret Sharing [76], and constructs an infrastructure for secure multiparty computation that provides: (a) flexibility, for servers to join and leave computations; (b) fault-tolerant properties, as t servers can fail (as crash or Byzantine faults) without interrupting the computation; (c) guaranteed output termination, meaning adversaries are not

able to stop honest servers from completing computations; (d) and malicious security, providing confidentiality and integrity of computations up to t faults as long as $n = 3t + 1$ total servers are assumed.

1.4 Goals

The primary objective of this work is to understand and resolve challenges regarding the limitations of current Multi-Party Computation (MPC) protocols which block their use in practical, long-term services.

The development process passed by the comprehensive literature on MPC, its respective state-of-the-art, and relevant cryptographic concepts such as verifiable and proactive secret sharing, zero-knowledge proofs, fault-tolerant mechanisms, and state-machine replication systems. The study of the COBRA library was also critical for understanding its functionality and defining an MPC application layer that could enable dynamic computations over stored client data under asynchronous networks. We then specified the models needed to develop a practical MPC system to fit the underlying BFT-SMR framework, defined our own dynamic MPC framework (MPCServe) based on such models, and implemented mechanisms to ensure dynamic and secure computations against Byzantine Adversaries with guaranteed output. Finally, we evaluated MPCServe under different configurations to analyze the impact of additions and multiplications over different circuit types.

1.5 Publications

The work developed in this dissertation resulted in the following publication:

Practical and Fault-Tolerant Secure Multi-Party Computation as a Service. Miguel Carvalho, Robin Vassantlal, Alysson Bessani, Bernardo Ferreira. Proceedings of the 15th Simpósio Nacional de Informática (INForum'24). Lisboa, Portugal. 2024.

1.6 Structure

The remainder of this work assumes the following structure:

- **Chapter 2** refers to the background information about core concepts of this thesis, such as secret sharing, multi-party computation, and distributed systems;
- **Chapter 3** discussing the current state-of-art of relevant topics surrounding multi-party computation;
- **Chapter 4** describes our approach for modeling practical MPC;
- **Chapter 5** and **Chapter 6** present our MPC framework for distributed computations over asynchronous networks assuming a passive and malicious adversary, respectively;

- **Chapter 7** presents experimental results of the system evaluation;
- **Chapter 8** points to future work and concludes this thesis.

Chapter 2

Background

This chapter provides context as we analyze and discuss the core concepts that are necessary to the understanding of our work.

2.1 Secret Sharing

Secret Sharing (Secret Sharing (SS)) is a cryptographic technique that allows a group of participants to hold a secret represented as unintelligible parts, commonly known as *shares*, such that reconstructing the original secret is possible by combining a sufficient number of distributed *shares*. The concept is formalized as a (t, n)-*threshold* secret sharing scheme, where t represents the number of shares needed to recover the secret and n corresponds to the total number of shares held by the group.

SS plays a crucial role as a tool to manipulate private information and finds usage in various applications such as threshold cryptography [33], Oblivious Transfer (Oblivious Transfer (OT)) [70] protocols [77], and distributed computations [26, 22].

2.1.1 Methods for Secret Sharing

Full-Threshold Secret Sharing. One way to define a secret-sharing scheme is to assume a Full-Threshold setting where the threshold t of shares required to obtain the secret is equal to the number of shares shared among n total participants (n = t). These schemes assume that all participants holding shares must collaborate to recover a secret, implying that no proper subset of participants can obtain information about the secret unless they are all present. Full-threshold schemes are useful in sensitive environments where each participant must be involved in the share reconstruction process, however, they are not resilient to faults and thus not viable in scenarios where participants may become unavailable or compromised.

Shamir's Secret Sharing. A widely known secret sharing scheme renowned for its flexible capabilities is *Shamir's* [76] scheme, which is an $(t + 1, n)$-*threshold* scheme based on polynomial interpolation in finite fields. Shamir offers a significant improvement over traditional Full-Threshold SS by enabling the dealer to share a secret s with n participants so that any subset of

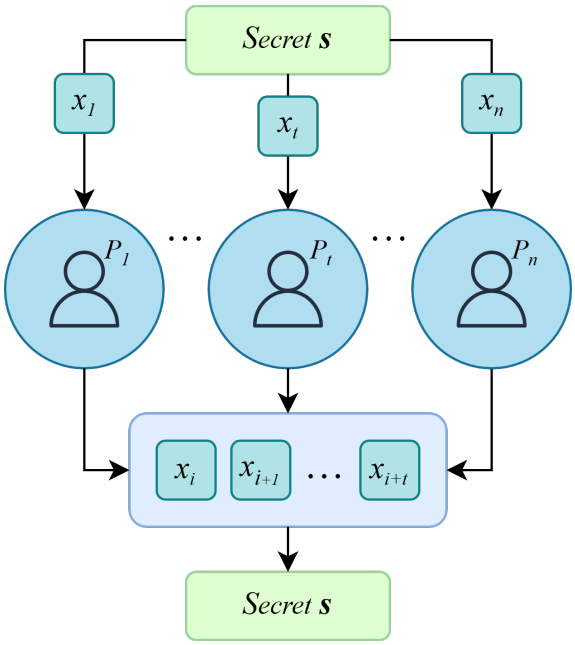

Figure 2.1: Shamir's Secret Sharing [76]. Assuming $n > t$, any subset of $t + 1$ shares allows for reconstructing the original secret s.

$t + 1$ shares can be used to reconstruct the original secret, allowing any combination of $t + 1$ participants to work together and obtain s.

Shamir's scheme represents *shares* of a chosen secret s as points of a degree-t polynomial $f(x) = c_t x^t + ... + c_1 x + s$, made from non-zero randomly generated coefficients $c_1, ..., c_t$ in field F except the constant term which holds the secret $f(0) = s$. It is then possible to reconstruct the secret s by defining a new polynomial $f_{new}(x)$ with constant term $f_{new}(0) = s$ which can be obtained through Lagrange Interpolation [80]:

$$f(0) = \sum_{j=0}^{t} y_j * \prod_{m=0, m\neq j}^{t} \frac{x_m}{x_m - x_j}$$

where (x_i, y_i) are any subset of $t + 1$ generated points/shares of $f(x)$. More formally, Shamir's Secret Sharing (Figure 2.1) is defined by:

- A *share* protocol, which receives a secret s (under finite field F) to be shared and the threshold t of shares required to combine the secret, and outputs shares $(f(1), ..., f(n))$, which are polynomial evaluations of a constructed degree-t polynomial $f(x)$ over t random coefficients $c_1, ..., c_t \in F$ which holds $f(0) = s$.

- A *reconstruct* protocol, which receives a list of shares and, assuming all shares lie on a single polynomial, attempts to find a degree-t polynomial $f_{new}(x)$ from $t + 1$ or more shares through polynomial interpolation. If there is such a polynomial, the protocol outputs the coefficients of this new polynomial, which can then be used to build $f_{new}(x)$ and obtain the original secret $f_{new}(0) = s$.

2.1.2 Verifiable, Proactive, and Dynamic Secret Sharing

A limitation of *Shamir's* scheme comes with its vulnerability against active adversaries since if any of the chosen $t+1$ shares for secret reconstruction becomes corrupted, the resulting interpolated polynomial may reveal a completely different secret.

Verifiable Secret Sharing (Verifiable Secret Sharing (VSS)). To provide security in this matter, an SS scheme can be enhanced with mechanisms to allow the verification of shares [23] through integrity proofs such as commitments. A commitment proves that shares were correctly generated, enabling secret sharing participants to confirm whether a given share has been manipulated. Non-interactive linear commitment schemes [45, 69] define their commitments based on the polynomial used for the secret sharing process. Feldman's [45] scheme follows an approach that relies on the coefficients of the underlying polynomial, defining *verifiable shares* as:

$$vs_a \stackrel{def}{=} \langle a, C_{f(x)} \rangle$$

where $C_{f(x)} \stackrel{def}{=} \langle C_0, ..., C_t \rangle$ are the Feldman commitments of a given share a lying on polynomial $f(x)$, where $C_j \stackrel{def}{=} g^{c_j}$ for $0 <= j <= t$ assuming g as a public generator of a cyclic group (which reveals no information under the discrete log hardness assumption). This way, secret-sharing participants can verify the integrity of a given share $a_j = f(j)$ by checking if the condition:

$$g^{f(j)} = \prod_{k=0}^{t} C_k^{j^k}$$

is true, confirming that share a_j is associated to the polynomial $f(x)$.

Proactive Secret Sharing (Proactive Secret Sharing (PSS)). *Proactive Secret Sharing* [68] provides secure secret sharing in long-lived systems by allowing the periodic refresh and recovery of participants shares. PSS was created to deal with a "mobile" adversary which, in contrast with other types of adversaries, may eventually compromise more than t shareholders throughout the secret sharing process. Such schemes [51] rely on a renewal polynomial $f_{renewal}$ and recovery polynomial $f_{recover}$ to respectively *refresh* and *recover* shares of a given polynomial $f(x)$, such that:

- $f_{renewal}$ *updates* shares of polynomial $f(x)$ and $f_{renewal}(i) + f(i)$ defines a new valid share;
- $f(i) \neq f_{renewal}(i) + f(i)$ is true, assuming $f_{renewal}(i) \neq 0$;
- $f(0) = f_{renewal}(0) + f(0)$ preserves the original secret $f(0)$, where $f_{renewal}(0) = 0$;
- Share $f(i)$ can be recovered by $f(i) = f_{recover}(i) + f(i)$, where $f_{recover}(i) = 0$.

Dynamic Proactive Secret Sharing. Dynamic Proactive Secret Sharing [75, 63] (Dynamic Proactive Secret Sharing (DPSS)) combines VSS and PSS to enable a flexible, secure environment with both share verification, recovery, and support for changes to the set of underlying secret-sharing participants, which is desirable for distributed environments where participant availability fluctuates.

2.2 Homomorphic Secret Sharing

Homomorphism refers to the mathematic mapping that preserves the nature of operations between two algebraic structures. Secret sharing schemes are said to be homomorphic [74] if they preserve the nature of operation f over a specific set of secrets when performing that same operation f over the shares that hide those secrets.

Addition Homomorphism. *Shamir's* scheme, by relying on polynomial interpolation, makes it inherently *additively* homomorphic, as adding two different polynomial secrets s and s' is mathematically equivalent to adding the coefficients/shares of their respective polynomials. More specifically, adding shares $f(i)$ and $h(i)$ from polynomials $f(x)$ and $h(x)$ will return a new addition share $k(i) = f(i)+h(i)$ linked to the polynomial $k(x) = f(x)+g(x)$ which represents the sum of the two underlying secrets. As a result, its polynomial evaluation $k(0)$ returns $f(0)+h(0) = s+s'$.

Similarly, the same homomorphic property can be applied to some linear commitment schemes. For example, by relying on the coefficients of Shamir's polynomials to define a *VSS* environment, Feldman's scheme [45] grants calculating the commitment to the addition share $k(i) = f(i)+h(i)$ by directly computing the product of the respective polynomial commitments $g^{f(x)}$ and $g^{h(x)}$:

$$C_{k(x)} = C_{f(x)}C_{h(x)} = \langle g^{c_0^{f(x)}+c_0^{h(x)}}, g^{c_1^{f(x)}+c_1^{h(x)}}, \ldots, g^{c_t^{f(x)}+c_t^{h(x)}} \rangle$$

as adding shares requires multiplying their respective commitments to operate over the exponents which refer to the coefficients of each involved polynomial.

Multiplication Homomorphism. While additive homomorphism is straightforward, achieving multiplications under a secret sharing environment presents exponential difficulty. When multiplying shares associated with degree t polynomials, the degree of the resulting polynomial consequently increases to $2t$, converting the needed share threshold to at least $2(t+1)$ so as to reconstruct the product of the original secrets. Multiplying commitments also presents additional complexity, as to compute the commitment to the product $k(i) = f(i) \cdot h(i)$ one would need to compute $g^{f(x)\cdot h(x)}$, which is not straightforward as it requires managing the product of two polynomials in the exponent.

Many environments bypass this issue by transforming the multiplication problem into one of computing linear equations over shares. For example, SPDZ [30] relies on generating a series of multiplication triples over randomized polynomials - called Beaver triples - to be used in a specific linear equation that defines correct multiplication shares. BGW [11] addresses the multiplication issue in a synchronous setting by first randomizing the degree-$2t$ polynomial so that it is uniformly distributed, then reducing its degree to the target threshold t. Furthermore, Gennaro et al. [46] achieves both the previous degree reduction and randomization step in a single action by exploring the fact that share interpolation leads to a paradigm in linear algebra amounting to the inversion of a Vandermonde [62] matrix. Mathematically speaking, it follows that a Vandermonde matrix of

size $(2t+1)$ by $(2t+1)$ is defined as:

$$V_{2t+1} = \begin{bmatrix} x_1^0 & x_1^1 & \dots & x_1^{2t} \\ x_2^0 & x_2^1 & \dots & x_2^{2t} \\ \vdots & \vdots & \ddots & \vdots \\ x_{2t+1}^0 & x_{2t+1}^1 & \dots & x_{2t+1}^{2t} \end{bmatrix}$$

where x_i are unique public identifiers associated with each participant. Then, assuming that each participant holds shares $a = f(i)$ and $b = g(i)$ of polynomials $f(x)$ and $g(x)$ of degree t, respectively, the product of the previous polynomials can be represented as $h(x) = f(x)g(x) = c_{2t}x^{2t} + ... + c_1x + ab$. It is then possible to infer that:

$$V_{2t+1} \begin{bmatrix} ab \\ c_1 \\ \vdots \\ c_{2t} \end{bmatrix} = \begin{bmatrix} h(1) \\ h(2) \\ \vdots \\ h(2t+1) \end{bmatrix}$$

Consequently, the previous statement implies that ab can be obtained using the first row of the inverse Vandermonde matrix $V_{2t+1}^{-1}[0]$ and the right-hand side of the equation, revealing $ab = \lambda_1 h(1) + ... + \lambda_{2t+1} h(2t+1)$. Finally, it is possible to set new and random polynomials $z_1(x), ..., z_{2t+1}(x)$ of degree t which satisfy $z_i(0) = h(i)$ such that $Z(x) = \sum_{i=1}^{2t+1} + ... + \lambda_i z_i(x)$ is a polynomial of degree t with constant term $Z(0) = \lambda_1 h(1) + ... + \lambda_{2t+1} h(2t+1) = ab$, making evaluation points $Z(i)$ of $Z(x)$ valid multiplication shares of correct degree t. Still, to operate under asynchronous environments, these protocols rely on asynchronous primitives such as Byzantine Agreement [58] (Byzantine Agreement (BA)) sub-protocols to enable participants to agree on the set of Beaver triples used during multiplications, as we later discuss.

2.3 Zero Knowledge Proofs

Interactive Zero-Knowledge Proofs. A Zero-Knowledge (Zero Knowledge (ZK)) proof [49] is an evidential procedure that allows one entity (the prover) to convince another (the verifier) that a given statement is true without revealing any additional information about the statement itself. Formally, a ZK proof is defined as a triplet $(\text{Setup}, \text{Prove}, \text{Verify})$ where the prover, given a statement ϕ and witness w, produces a proof ϕ such that $\pi = \text{Prove}(CRS, \phi, w)$, which can be verified by computing $\text{Verify}(CRS, \phi, \pi)$ while also satisfying:

- Completeness, meaning if the statement is true, then an honest verifier can be convinced by an honest prover of the statement's validity;
- Soundness, meaning if the statement is false, no dishonest prover can convince the verifier that it is true (except with a negligible probability);
- Zero-Knowledge, meaning if the statement is true, the verifier learns nothing beyond the fact that the statement is true (no additional information about the prover's secret is revealed by the proof).

Under a secret-sharing setting, ZK proofs have been continuously sought after [78, 48], as commitments used in verifiable secret-sharing schemes cannot alone guarantee that participants act honestly during computations or correctly form products of shares. As such, ZK proofs have been developed to augment VSS by enabling participants to prove that the product of two shares, $ab = a * b$, has been correctly computed, ensuring that any malicious user either shares the correct product or does nothing harmful.

Non-interactive Zero-Knowledge Proof. Non-interactive Zero-Knowledge (Non Interactive Zero Knowledge (NIZK)) [16] proofs extend standard ZKPs by defining a variant where the proof is verified with reduced interaction between the prover and the verifier. A direct approach to transforming ZKPs into NIZKPs is through Fiat-Shamir's heuristic [13], which removes the need for a 3-round communication protocol by substituting the proof challenges with a hash function under the random oracle model.

Improving the efficiency of ZKPs is crucial in secret sharing, as the number of required proofs and communication rounds scales linearly with the number of secrets; therefore, minimizing communication overhead is essential for maintaining protocol efficiency.

2.4 Multi-Party Computation

2.4.1 Definition

Multi-Party Computation (MPC) [82] is a family of protocols that allow a group of distributed participants, who may not trust each other, to securely compute an agreed-upon computation over their private, encrypted inputs. MPC systems assume an initial setup where n participants P_i hold a piece of sensitive data $secret_i$ (only known to that user), allowing parties to collaborate and compute a publicly known function $f(secret_1, ..., secret_n)$ over their secrets while ensuring *Correctness*, where the computational output is correctly computed even in the presence of corruption, and *Privacy*, where the computational output is the only new information publicly released to the participants.

Security. MPC defines its security under a given corruption number, adversarial behavior, and termination guarantee, setting the system's capabilities for a considered environment.

- *Corruption Number* tracks the number of corrupted parties (t) by separating the disruptive aspect into (a) Honest Majority, where the number of honest parties outweighs the number of corrupt parties ($n/2 > t$), and (b) Dishonest Majority, where the number of corrupt parties may exceed the number of honest parties ($n > t$).

- *Adversarial Model* classifies adversaries depending on their corrupt nature. An adversary can be presented as (a) Semi-Honest, where adversaries follow protocol execution but may obtain state information held by the corrupted party; or (b) Malicious, where adversaries may take total control of the corrupted party and arbitrarily deviate from the intended protocol as means to compromise security and privacy. Additionally, we may define adversaries as (a) Static, where the set of corrupted parties is fixed before the execution starts; or (b)

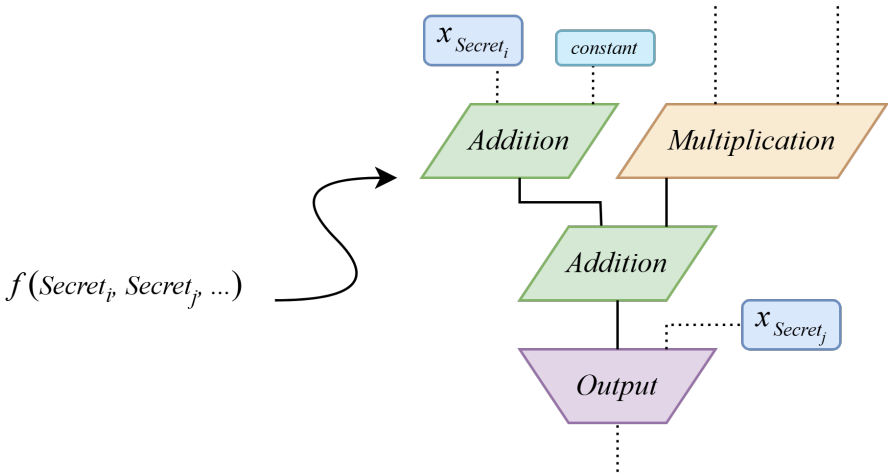

Figure 2.2: Example of function-to-circuit conversion. A circuit operates over shares of secrets, where each gate could receive input shares, constant values, or outputs of other gates.

Adaptive, where adversaries may choose to corrupt different parties during execution as long as the threshold of total corrupted parties t is maintained.

- *Termination Guarantee* defines how MPC computations are reliably concluded, splitting criteria into (a) Fairness, which ensures that corrupted parties get the output only if all honest parties also receive it; (b) Secure-with-abort, where corrupted parties are allowed to learn the output before the honest parties, but honest parties can choose to abort based on the corrupted parties actions; or (c) Guaranteed Output, the strongest guarantee where malicious adversaries cannot prevent honest participants from getting computational output.

Circuits. Multi-Party Computation defines an environment for evaluating any computation as long as it can be expressed as a series of boolean or arithmetic operations (Figure 2.2) and the respective MPC protocol supports them. These circuits are constructed in such a way to allow their distribution amongst each MPC participant who evaluates it in a gate-by-gate fashion over their private inputs.

The first formal protocol was introduced in 1986 by Andrew Yao [82], who constructed a secure two-party computation protocol by transforming Boolean functions into Garbled Circuits and sharing/validating the circuit amongst participants without gaining access to the original inputs. Alongside BMR [7], an extension of Yao's protocol designed for settings with more than two parties, these protocols paved the way for more complex MPC frameworks which are resorted to in a variety of privacy-preserving distributed applications, ranging from simple arithmetic operations [65] [11] to more advanced tasks such as Statistical Analysis [37], Data Mining [17, 59], and Machine Learning [60, 57].

2.4.2 Secret Sharing-based Multi-Party Computation

A natural way to construct MPC protocols comes by leveraging Secret Sharing, as some SS schemes (including Shamir's) allow operation over polynomial shares of secrets from differ-

ent shareholders. Unlike heavy-dependent computational approaches that rely on Fully Homomorphic Encryption [2] (Fully Homomorphic Encryption (FHE)), SS-based MPC achieves distributed computations by allowing direct (or almost) manipulation of user shares and offers flexibility by enabling functionality under a threshold of participants. This concept was explored by Goldreich, Micali, and Wigderson (GMW) [65], who generalized Yao's Garbled Circuits [82] to more than two parties and proposed Multi-Party Computation based on Full-Threshold SS. Later, Goldwasser, Ben-Or, and Wigderson (BGW) [11] introduced a protocol based on Shamir's scheme, enabling parties to obtain the output of computations using a threshold t number of shares. SPDZ [30] extended these ideas and defined a protocol based on a computationally heavy offline phase for Beaver Triple [6] generation to provide efficient computations up to $n - 1$ corruptions under abort termination guarantees.

MPC Framework	**Nº of Parties**	**Circuit Type**	**Corruption Number**	**Adversarial Model**	**Termination Guarantee**
Yao's GC [82]	2	Boolean	Dishonest	Passive	Fair
GMW [65]	2+	Arithmetic	Dishonest	Passive	Secure-with-Abort
BGW [11]	2+	Arithmetic	Honest	Passive	Guaranteed Output
BMR [7]	2+	Boolean	Dishonest	Active	Secure-with-Abort
SPDZ [30]	2+	Arithmetic	Dishonest	Active	Secure-with-Abort

Table 2.1: Foundational MPC frameworks.

While SS-based protocols effectively handle addition operations locally among participants, the complexity of multiplication is often addressed in ways that impede the system's capabilities concerning communication assumptions and participant failure management, as we will discuss later.

2.5 State Machine Replication

Distributed systems assume multiple interconnected nodes that operate autonomously and coordinate through a shared network to provide a predefined set of services masked as a single coherent system. State Machine Replication (State Machine Replication (SMR)) [73] is a fundamental approach that allows replicas of a distributed system to maintain consistent states, enabling fault-tolerant and highly available services.

2.5.1 Methods for State Machine Replication

Byzantine Fault Tolerant State Machine Replication (Byzantine Fault Tolerant State Machine Replication (BFT SMR)). Under a real-world environment, SMR systems are naturally susceptible to several types of failures, including network issues, crashes, and attacks on the infrastructure. To enhance fault tolerance, Byzantine Fault-Tolerant SMR (BFT SMR) extends tra-

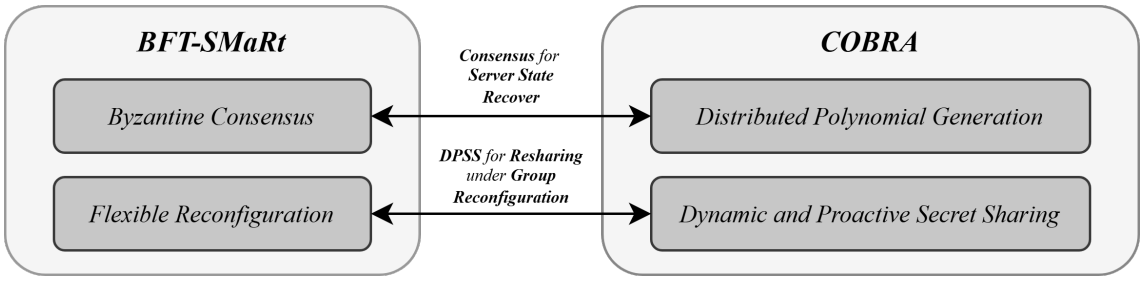

Figure 2.3: COBRA protocol stack.

ditional SMR by tolerating up to t Byzantine failures amongst $n \geq 3t + 1$ total replicas, where faulty replicas can exhibit arbitrary, and potentially malicious behavior. A prominent and robust BFT SMR implementation comes with BFT-SMaRt [14], which was the *first* BFT SMR library to support the dynamic reconfiguration of the SMR replica set, defining support for transparent and durable services in asynchronous environments.

Confidential BFT SMR. Still, guaranteeing confidentiality in SMR to achieve secure distributed computations is notoriously complex. Assuming the client-server model, confidentiality in SMR requires that data privacy is ensured during client-server interactions and within the replicated service. One way to achieve confidentiality in SMR is through threshold cryptography [34], which ensures that data remains secure and inaccessible even if a threshold number of BFT SMR replicas becomes compromised.

Building on this model, the COBRA [79] solution (Figure 2.3) defines confidential BFT SMR for privacy-preserving applications by leveraging consensus and dynamic, proactive secret sharing mechanisms to securely distribute polynomial shares of client data, which are maintained by a set of servers while ensuring that no individual server has access to the complete data.

2.5.2 Discussion

BFT SMR relation to MPC. Under a *Multi-Party Computation* environment, BFT-SMaRt [14] may provide an underlying infrastructure to support the dynamic and reliable execution of circuit evaluations by ensuring that, despite the presence of Byzantine faults, the nodes can securely and efficiently perform computations on private inputs while maintaining the integrity and confidentiality of the computational process. Also, the topic of party replaceability - typically discussed in the context of Byzantine agreement [58] and state machine replication - is highly relevant to the setting of MPC, as it would allow systems to adapt to changes in participation without interrupting computations. We see BFT-SMaRt [14] addressing this by enabling replicas to be added or removed *on-the-fly*, allowing for dynamic participant set reconfiguration in contrast to other BFT SMR systems that only assume a fixed set of replicas. As such, we believe that an MPC framework grounded in a BFT SMR model is the most effective approach to achieve *true* practical and guaranteed output MPC against a malicious adversary.

COBRA relation to MPC. Consequently, by leveraging BFT-SMaRt [14], COBRA [79] presents a compelling framework to establish a secure environment for *Practical Multi-Party*

Computation. COBRA enhances traditional secret-sharing through its Dynamic Proactive Secret Sharing (DPSS) [75] scheme that allows verification, recovery, and refreshing of private shares, ensuring client data confidentiality against mobile adversaries over extended periods. Additionally, by following a client-server architecture where each server maintains a Key-Value (Key-Value (KV)) store of client data shares, information is maintained and retrieved with integrity. It also enhances availability by simplifying access and integration with cloud services and enables secure state transfer during player churn, possible from the underlying BFT SMR layer. This way, we see relevance in defining a flexible, fault-tolerant MPC protocol that extends COBRA to enable computations over stored shares maintained by COBRA's KV store while incorporating a reliable and dynamic environment over a practical BFT SMR protocol that helps offloading computation from traditional MPC participants to a set of flexible *work* servers.

Chapter 3

Related Work

Context. As data privacy regulations become more stringent and the demand for secure data sharing increases, a core challenge comes with making Multi-Party Computation (MPC) practical solutions for real-world usage by businesses. In this chapter, we explore the current state of the art regarding Generic-Purpose (and more popular) MPC frameworks, Proactive MPC, MPC in the Cloud, Dynamic MPC, and Asynchronous MPC. By following this line of presentation, we aim to show how current frameworks present crucial limitations that fail to address the requirements imposed by realistic operational environments, namely assuming fixed participants that are required to stay the entire computation, following synchronous networks, not naturally handling scenarios where parties fail to communicate with each other, and aborting without securing any computational result.

3.1 Generic-Purpose MPC

Generic-Purpose MPC. Several widely recognized MPC frameworks gained popularity due to their ease of use and versatility in enabling generic-purpose computations. For example, FairplayMP [9] and MOTION [19] proposed frameworks built upon well-established protocols, utilizing a rearranged BMR [7] protocol with an honest-majority BGW [11]-style setup phase and a version of the GMW [65] protocol combined with an Oblivious-Transfer-based BMR [7], respectively. Additionally, MP-SPDZ [55] offers an environment to benchmark a wide range of security models, supporting 34 protocol variants that extend SPDZ-2 [56], an implementation of the foundational SPDZ [30] protocol. SCALE-MAMBA [1], similarly, adopts an SPDZ-based MPC approach, implementing a maliciously secure hybrid protocol based on a low-level compiler in the MAMBA language and introduces SCALE, a system that enhances the actively secure SPDZ [56] framework with improved online and offline phases.

Despite their contributions, these frameworks face limitations that hinder their practical applicability. Key issues include operating in synchronous environments, not supporting dynamic participation during computation, providing only security-with-abort guarantees (where computations may terminate prematurely without producing results), and reliance on non-optimized protocols. Framework-specific challenges also involve: Fairplay [9] supporting only semi-honest adversaries;

MPC Framework	**Corruption Number**	**Adversarial Model**	**Termination Guarantee**	**Service Model**	**Dynamic?**	**Non-Sync.?**
EOPY [41]	Honest	Active	w/Abort	Traditional	No	No
ELL [40]	Dishonest	Active	w/Abort	Traditional	No	No
EasyCrypt [42]	Honest	Passive	Guaranteed	Traditional	No	No
WWSY [81]	Honest	Passive	w/Abort	Client/Server	No	No
LTV [61]	Honest	Active	w/Abort	Client/Server	No	No
MPCSaaS [3]	Honest	Active	w/Abort	Client/Server	No	No
Fluid [24]	Honest	Passive	w/Abort	Fluid	Yes	No
YOSO [47]	Honest	Passive	Guaranteed*	Fluid	Yes	No
LeMans [72]	Dishonest	Passive	w/Abort	Fluid	Yes	No
Max. Fluid [32]	Honest	Active	Guaranteed	Fluid	Yes	No
HNP [52]	Honest	Active	w/Abort	Traditional	No	Yes
FIN [38]	Honest	Active	Guaranteed	Traditional	No	Yes
MPCServe	Honest	Active	Guaranteed	Client/Server	Yes	Yes

Table 3.1: MPC-related frameworks vs Our proposed framework (MPCServe). The Fluid Model assumes an underlying Client/Server model. *Only Information-Theoretic

MOTION [19] suffering from high communication complexity ($O(n^4)$), making large-scale computations impractical; SCALE-MAMBA [1] struggling with cryptographic parameter complexity and limited computation models; and MP-SPDZ [55] presenting performance bottlenecks due to local deployment assumptions.

3.2 Proactive, Cloud, and Dynamic MPC

Proactive MPC. Proactive MPC assumes a line of work where a set of MPC parties continuously have their internal state updated due to an adversary that may attempt to corrupt each participant throughout the computational effort. While current frameworks [41, 40, 42] provide refreshing mechanisms to allow data confidentiality for long periods, they often overlook important aspects regarding system capability. One example comes with EasyCrypt [42], which defines a protocol based on the BGW protocol for building and verifying high-confidence cryptographic proofs under static security guarantees with passive adversaries. Akin to this, other protocols [41, 40] define simple proactive computational environments but inherit issues typical of generic-purpose MPC, namely relying on a fixed participant set, synchronous channels, and computationally intensive subroutines to ensure security. Consequently, while proactive mechanisms are essential for long-term security in MPC, they are insufficient to ensure practical and scalable computations.

Cloud MPC. Cloud Multi-Party Computation protocols have also been developed to give MPC availability and compatibility over cloud services [81, 61, 3]. These frameworks redefine

the traditional MPC structure by following a model where participants are split into the client domain, who initiate computation requests and provide inputs, and the server domain, responsible for interpreting and performing requested operations.

For example, [81] defines an MPC-as-a-service protocol for generic purposes supporting at most $n - 1$ malicious clients by migration computations to a core server. A more complex and interesting approach comes with [61], which defines an *on-the-fly* multiparty computation through a multi-key Fully-Homomorphic Encryption (FHE) [2] scheme that operates over client inputs. Despite following a design that offers efficiency improvements by offloading computational tasks to a given service, these protocols rely on a single, possibly vulnerable semi-honest server. This reliance represents a weak assumption in cloud environments, as it introduces a single point of failure in the system's security model. Furthermore, the so-called "on-the-fly" dynamism referred to in many Cloud frameworks [61, 3] is misleading, as it only applies to client space, while the underlying server architecture remains static. In the practical setting, we are interested in MPC which offers general flexibility by allowing servers to exit and re-enter the computational process.

Dynamic MPC. *Dynamic MPC* refers to the domain of MPC protocols that take party replaceability as their core priority. Such protocols focus on the dynamic joining and leaving of parties during computations while achieving "maximal fluidity", where each participant only needs minimal involvement (one round of communication) to contribute to the computation. The Fluid model was first introduced in [24], which unlike traditional client-server MPC where servers are statically assigned, allowed the dynamic joining and leaving of servers during computation. The model follows a layered-style methodology where computations are divided into a sequence of epochs, wherein each epoch a designated committee of servers works together to compute gates of the circuit and ensure future committees have the necessary state to ensure computational progress. Security in Fluid MPC is achieved by associating each gate's wire values with a secret sharing of a Message Authentication Code (Message Authentication Code (MAC)) [35], whose shares are incrementally processed throughout protocol execution to allow verifying computational correctness at the output stage.

Several efforts were motivated by this flexible MPC model. Le Mans MPC [72] extends Fluid MPC and proposes an SPDZ-based methodology to support a dishonest majority security setting by defining a universal preprocessing phase to generate randomness and reduce the computational effort of committees during the online computation phase. YOSO [47] presents a statistically secure, public-key-based MPC protocol with output termination guarantees, focusing on participant role assignment to achieve fluidity in the MPC setting. Additionally, Maximally-Fluid MPC [32] introduces a perfectly secure, guaranteed output Fluid-style MPC by building on an optimized version of the BGW protocol [11] combined with homomorphic commitments, enabling verifiable secret sharing and maximal fluid computations where each committee may only execute one round of communication per epoch.

Still, a major critique of these Fluid-based frameworks lies in their reliance on a synchronous communication environment that assumes the orderly, round-based execution of operations by its

dynamic committees. In an asynchronous setting, network delays can disrupt the timing required for predictable round completion, a critical issue that is not addressed by such frameworks. Additionally, many of these frameworks consider only malicious adversaries capable of manipulating distributed shares and observing participant states and disregard the existence of a Byzantine adversary who beyond this may cause message delivery failures or force participants to crash. Moreover, we have that: Fluid MPC [24] and Le Mans MPC [72] are limited to security-with-abort termination guarantees; YOSO [47] needs strong computational and trusted setup assumptions while being only statistically secure if the adversary corrupts a *random* subset of parties; and both YOSO [47] and Maximally-Fluid MPC [32] depend on *secure channels to the future* to broadcast messages to committees that are expected to participate later in the computational effort to ensure that secret sharing is securely conducted.

3.3 Asynchronous MPC

Asynchronous MPC. *Asynchronous MPC* describes frameworks that aim to provide secure computations over asynchronous networks. The study of asynchronous multi-party computation focuses on the fact that in such environments, the computational capacity of the network is impacted by a byzantine adversary that beyond distributing malicious shares can withhold necessary information during critical stages, namely during the input stage (when distributing data shares for prepping computations) or during the computational process. Consequently, the integrity of asynchronous MPC relies on inputs from at least $n - t$ parties, as up to t parties may fail to participate. Because of this, a central challenge comes by addressing the *Agreement on a Core Set* (Agreement on a Core Set (ACS)) problem, where participants must agree on a common *core* set of at least $n - t$ parties whose broadcasts have been reliably completed. This way, if more than $n - t$ broadcasts are globally concluded, the ACS protocol ensures that all honest parties consistently identify the same core set. Additionally, ACS helps Beaver Triple generation in SPDZ [30]-based frameworks by identifying a core set of $n - t$ parties who complete Beaver triple sharing, and sub-share selection in BGW [11]-based frameworks where ACS identifies a consistent set of $n - t \geq 2t + 1$ distributed sub-shares to perform the final linear combination required for obtaining multiplication shares with the correct degree. Asynchronous MPC was first introduced in [10], which generalized the synchronous BGW [11] protocol and incorporated primitives to achieve asynchronous MPC in a $t < n/4$ security setting tolerating t Byzantine adversaries. Subsequent research reduced the corruption threshold to $t < n/3$ [31, 52, 25], further advancing the feasibility of asynchronous computation. More recently, some works [38, 84] have achieved asynchronous MPC by combining Asynchronous Reliable Broadcast (ARB) [18] with Multi-Valued Validated Byzantine Agreement (MVBA) [20] to address ACS.

Still, asynchronous MPC frameworks present some limitations. As previously mentioned, a core issue lies in resolving ACS, which typically depends on Byzantine agreement sub-protocols [10, 52, 38, 84] to achieve consensus on proposed sets. However, in optimistic scenarios where Byzantine interference may become temporarily absent, running a consensus protocol for every multipli-

cation can significantly slow down the overall performance of the framework. Furthermore, these frameworks primarily focus on correctness and output guarantees under Byzantine faults, but overlook the dynamic aspect of participants rejoining the computation and needing to construct their states through other participants. Additionally, we have that: most frameworks [52, 25, 38, 84] achieve $O(n^3)$ communication complexity, which is not optimal; [52] uses signatures to bypass implementing verifiable secret sharing mechanisms and disregards the capability of malicious servers manipulating the correctness of shares distributed during computation; [25] achieves asynchronous MPC only for boolean circuits; and [84] relies on server committees to generate Beaver Triples for subsequent committees during the computational effort, impacting protocol performance.

Chapter 4

Models for Practical Multi-Party Computation

This chapter focuses on expressing models for our dynamic multi-party computation setting. We define core concepts regarding circuits and their evaluation under an MPC environment, present the system, adversarial, and service model for our practical multi-party computation strategy, and also refer to security goals regarding our dynamic MPC model.

4.1 Circuit Definition

In this section, we present the concept of circuits used in the context of multi-party computational frameworks to represent requested functionalities.

Circuit. Similarly to novel MPC approaches, we assume that functions under the multi-party computation setting can be represented as an arithmetic circuit composed of addition and multiplication gates. An arithmetic circuit is interpreted as a directed acyclic graph of depth d and x operational gates, where each gate is characterized by two input wires, a basic arithmetic operation, and a single output wire. An arithmetic circuit extends to a *layered circuit* if it can be decomposed into well-defined layers, such that the output wires of gates on layer l can be fed as input to gates at the following layer $l + 1$ or any subsequent layers. More formally, an arithmetic circuit is said to be a layered circuit if:

- The circuit can be decomposed into layers such that each layer defines a gate to be evaluated and the output wire of gates at layer l can be used as input to gates at layer $l + 1$ or any subsequent layers;
- The circuit begins at layer $l = 0$ with an *input* gate that gets its input wires from an external environment (the client) and ends at layer $l = d + 1$ through an *output* gate that gets its inputs from the output wires of gates at layer $l = d$ or previous layers;
- The circuit is non-empty and contains one or more gates of the following types:
  - *Addition-by-Constant Gate*: Given two inputs $x, c \in F$, where c is a constant hard-wired to the gate, addition-by-constant gates output $z = x + c$;

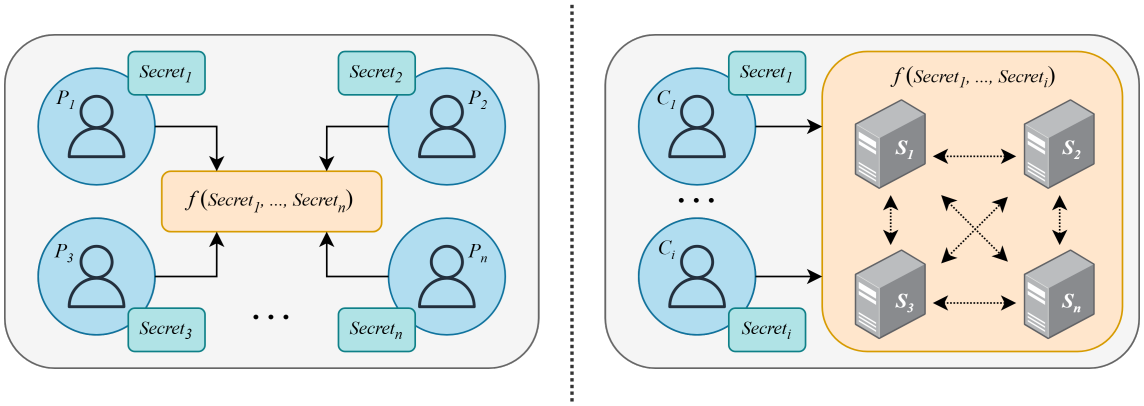

Figure 4.1: Traditional MPC (left) vs MPCServe's client-server setting (right). MPCServe replaces the need for participants to participate in computation by migrating work to a set of replicas that compute a given functionality over client data.

- *Addition Gate*: Given two inputs $x, y \in F$, addition gates output $z = x + y$.
- *Multiplication-by-Constant Gate*: Given two inputs $x, c \in F$, where c is a constant hardwired to the gate, addition-by-constant gates output $z = xc$;
- *Multiplication Gate*: Given two inputs $x, y \in F$, multiplication gates output $z = xy$;

This combination of addition and multiplication gates is sufficient and necessary for efficiently representing all possible MPC computations without needing other operation types. This is because subtractions can be expressed as the addition of the additive inverse, and divisions can be reframed as the multiplication of the multiplicative inverse (assuming a non-zero divisor). Converting a traditional circuit into a *layered circuit* is essential for structural and sequential evaluation of functionalities, as general arithmetic circuits do not inherently impose a clear order of gate evaluation or ensure that dependencies between gates are explicitly respected. As such, layered circuits guarantee that each MPC participant, holding a copy of the circuit, computes the same sequence of gates even in the context of an asynchronous network. We refer to layered circuits in the context of our multi-party computational protocol.

4.2 System Model

We consider a fully connected distributed system composed of a set of processes Π divided into two non-overlapping subsets: a set of n servers/replicas $\Sigma = \{s_1, s_2, ..., s_n\}$, and an unbounded set of clients $\Gamma = \{c_1, c_2, ...\}$.

Operations. We build upon COBRA's Confidential Key-Value (KV) Store, where requests are handled as a chronological sequence of atomic operations, and extend its interface for accessing/storing data by defining a new operation to allow secure computations on stored data. More specifically, clients access the multi-party computational system by requesting operations $O_p = \{Put, Get, Compute\}$ to the set of servers Σ, where:

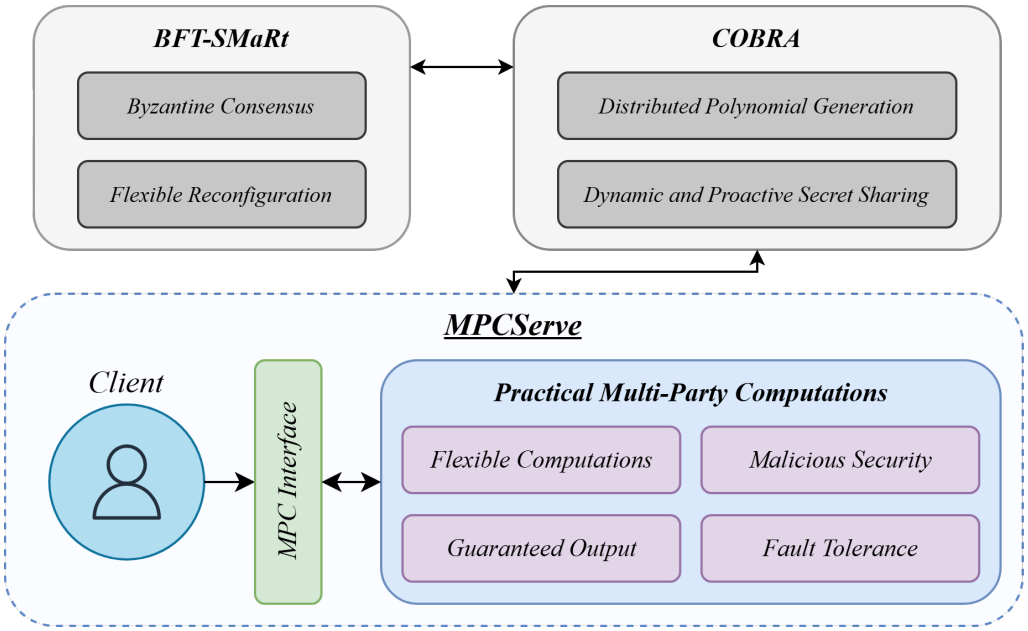

Figure 4.2: MPCServe's protocol stack. Purple boxes represent MPCServe's properties.

- Put defines a call for the COBRA's VSS $Share$ protocol to generate shares $a_i \in n$ (and respective commitment c) of a random degree t polynomial $p(x)$ and send each share a_i to the respective server in Σ;
- Get defines a call for the COBRA's VSS $Reconstruct$ protocol to retrieve secret $y = p(x)$, where $p(x)$ is a degree t polynomial interpolated using a set of $t+1$ valid shares maintained by servers in Σ;
- $Compute$ defines a call for our new computational protocol to *dynamically* evaluate a requested function f over shares of client data maintained by Σ.

Computational Model. The system follows a layer-by-layer gate evaluation methodology where servers evaluate gates of client-requested functionalities represented as layered circuits $L_c = \{g_1, ..., g_x\}$ of depth x over a sufficiently large finite field defined by COBRA's DPSS scheme. Computation is carried out by servers in Σ as a set of epochs $E = \{e_1, e_2, ..., e_{last}\}$ (e_{last} corresponds to the last epoch associated with a given computation), wherein each epoch assumes a committee $\Sigma^{e_i} \subseteq \Sigma$ that participates in evaluating a subset of gates of the circuit in a non-synchronous fashion. Committees between epochs may be subject to change, as we consider a dynamic system where servers can join and leave the system during the execution of a computation. Additionally, the composition of committees depends on the *view* of COBRA at the time of computation, where the most recent committee refers to the most up-to-date view of the system V_{cur}.

Communication. We assume a partially synchronous model [39] in which the network and processes may behave asynchronously until some *unknown* Global Stabilization Time (GST) after which the system becomes synchronous, with *known time bounds for computation and communi-*

cation. Also, every pair of processes communicates through private and authenticated *fair links*, i.e., channels guarantee the eventual delivery of messages without tampering or loss.

We also differ from the Fluid model [24], which depends on communication between different committees, by only requiring communication within the same computational committee. By following a BFT SMR approach where committees are tightly bound to COBRA's current view, the composition of each committee reflects the system's most up-to-date understanding of active servers. Consequently, if a committee experiences changes to its participant set, such as servers joining or leaving, the system automatically forms a new committee to reflect these updates. This way, we achieve true fluidity in the MPC setting, allowing for the dynamic and on-the-fly reconfiguration of committees during computation.

Moreover, in contrast with fully synchronous systems where computation progresses in well-defined rounds and each server waits for all others to complete a gate before moving to the next one, computation in MPCServe does not adhere to strict rounds, resulting in servers potentially operating on gates at different layers of the circuit concurrently. Nevertheless, due to the need for inter-server communication in multiplication gates (as we discuss later), the system requires that at least $2t + 1$ servers compute a multiplication gate at the same layer before advancing. Despite this requirement, computations under MPCServe follow a constant number of rounds/epochs, with the round complexity scaling linearly with the depth of the respective circuit.

Cryptographic Primitives. We rely on a trusted setup where each client/server presents a unique public identifier that can be verified through a *public key infrastructure*. As such, each server s_i owns a public-private key $\langle PubKey_i, PriKey_i \rangle$ for message signatures and encryption purposes. Additionally, we use the *SHA256* [67] hash algorithm and *AES* [29] with 256-bit keys to encrypt shares carried in inter-server exchanged messages to achieve secure communication and data confidentiality.

4.3 Adversarial Model

Our computational model addresses a *probabilistic polynomial-time (Probabilistic Polynomial-Time (PPT))* adaptive adversary capable of controlling the network and actively compromising up to a fraction $t < n/3$ of the servers within the current view V_{cur}. Due to the underlying COBRA library, we assume clients that access the infrastructure to be honest, meaning that they may only fail by crashing.

We render a *fully* malicious security setting where servers can at *any time* arbitrarily deviate from the protocol, i.e., they are prone to crash or Byzantine failures. In this setting, servers that do not follow the intended protocol are categorized as faulty or malicious, whereas those that operate correctly are deemed honest or correct. Additionally, the inner state of servers that become maliciously corrupted can be accessed and compromised by the adversary.

We also note that the traditional Fluid model [24] assumes each epoch of computation to consist of k rounds of communication under the same committee. In contrast, our solution follows a player elimination technique similar to previous work [64] which involves detecting and removing

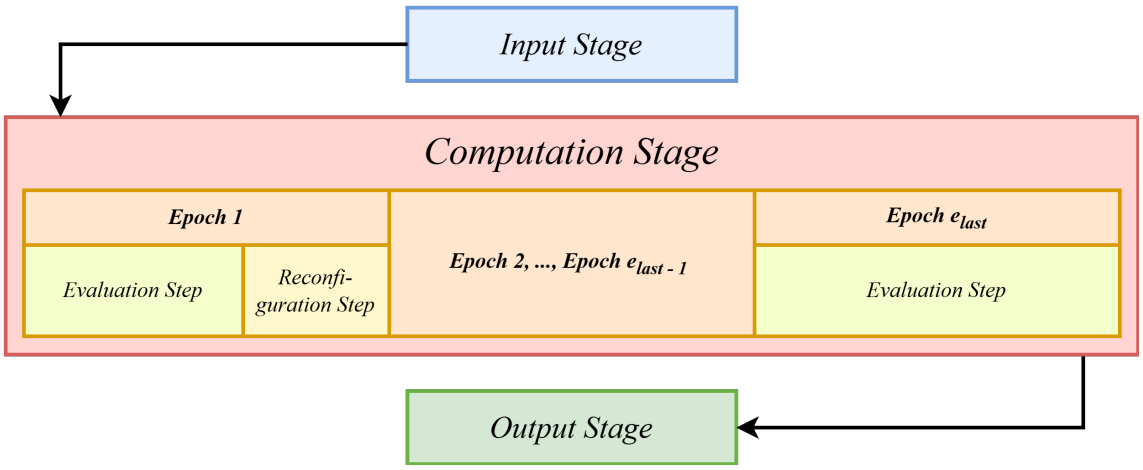

Figure 4.3: MPCServe's Service Model. Client inputs are fed to the Input Stage (Blue Box), which are sent to the Computation Stage (Orange Box) for evaluation of a given circuit, and results from computation are obtained through the Output Stage (Green Box).

malicious replicas *on-the-fly* so that they can no longer prevent the undermining of computations, impacting the constitution of committees during protocol execution. This approach relies on the fact that, even as malicious servers are identified and removed, they can be replaced by other servers to preserve the $t < n/3$ corruption ratio and maintain protocol security.

4.4 Service Model

We present MPCServe as a Multi-Party Computation framework that runs over COBRA's confidential state machine replication, following a client-server model where clients are capable of requesting operations $O_p = \{Put, Get, Compute\}$ to a set of dynamic servers that are subject to configuration changes.

Structure. The service model of MPCServe is composed of three main stages: (a) an input stage, where clients $c_i \in \Gamma$ hand their private data as shares to the committee of servers; (b) a computation stage, where servers participate in evaluating a layered circuit representation C of a client requested function f; and (c) an output stage, where clients learn of the computation's output by combining resulting shares from the circuit evaluation. We provide flexibility during the protocol execution as we model dynamism in the computation stage rather than the input and output stage. We now formally detail each core stage of MPCServe's service model.

Input stage. In the *input stage*, clients $c_i \in \Gamma$ request to evaluate a chosen function f, represented as a layered circuit C, by sending $O_p = Compute$ to the set of dynamic servers in Σ. Here, each server receives a *copy* of the representative circuit C, which must be well-defined and have its size proportional to the number of gates, to perform computations over their local states (maintained as data shares) while preserving computational privacy. We assume that operations $O_p = Compute$ are invoked only after the associated values have been previously stored in COBRA's Key-Value Store through $O_p = Put$ calls. Only when a valid circuit has been handed off

to the servers the execution progresses to the next stage.

Computation stage. In the *computation stage*, servers $s_i \in \Sigma$ interpret requests from clients and can evaluate requested function f, represented as a layered circuit C, over shares of data maintained by COBRA's SMR. This operation only proceeds if the designated stored data correlates to the required inputs of the circuit C. As mentioned before, the computational process evolves through a set of epochs $E = \{e_1, e_2, ..., e_{last}\}$, where e_{last} refers to the last epoch associated with a given computation. An epoch e_i only transitions to the following epoch e_{i+1} when the system suffers changes to the committee $\Sigma^{e_i} \subseteq \Sigma$ by addition or removal of servers, resulting in the new committee $\Sigma^{e_{i+1}} \subseteq \Sigma$ of epoch e_{i+1}. Each epoch e_i is further divided into two main steps:

- An *evaluation step*, where each server $s_i \in \Sigma^{e_i}$ evaluates one or more gates of the function's circuit representation by locally operating on stored shares or exchanging sub-shares between servers, depending on the gate type. This step concludes when $2t + 1$ servers have successfully evaluated the respective gates and generated their output as a series of valid shares to be used in future evaluation steps.

- A *reconfiguration step*, initiated when a new server s_{new} requests to join the current committee $\Sigma^{e_i} \subseteq \Sigma$ (which is the current set of servers responsible for evaluating gates of circuit C) of epoch e_i. We state that only servers from *outside* the committee Σ^{e_i} can announce their desire to join and aid the computational process. Depending on how far behind the new server $s_{new} \notin \Sigma^{e_i}$ is compared to others $s_i \in \Sigma^{e_i}$, the joining server uses *recovery and re-sharing* mechanisms from COBRA's DPSS [79], or resorts to COBRA's SMR Durability Techniques [14] (namely Logs and Snapshots) to obtain valid shares of private data and continue computation without causing the system to halt. After a state transfer step has completed for committee Σ^{e_i} in epoch e_i, the system progresses to a new epoch e_{i+1}, establishing a new committee $\Sigma^{e_{i+1}}$ which includes the new server s_{new}.

Output stage. In the *output stage*, the committee $\Sigma^{e_{last}}$ of the last epoch e_{last}, which corresponds to the set of servers that have correctly evaluated the last gate of circuit C, returns the output of the requested operation $O_p = Compute$ in the form of shares, which are delivered to the client and combined as a final result. This stage determines the end of an execution instance where a sequence of committees $\Sigma^{e_1}, \Sigma^{e_2}, ..., \Sigma^{e_{last}}$ successfully computed a layered circuit C representation of function f requested by a client.

4.5 Security Goals

MPCServe aims to ensure **Correctness** by leveraging COBRA's fault-tolerance properties and implementing mechanisms to handle server failures caused by crashes or network delays and malicious attempts to send invalid data (e.g., corrupt shares).

Furthermore, our solution seeks to provide **Privacy** by utilizing COBRA's Dynamic Proactive Secret Sharing (DPSS) to securely distribute and continuously refresh shares of client data maintained by servers during computations. By following this approach, we can take advantage of the

additive-homomorphic property of COBRA's DPSS which grants the capability to operate directly over shares maintained by servers without revealing sensitive client data. Also, the verification and resharing mechanism of COBRA's DPSS blocks the ability of a fully malicious adversary to reconstruct confidential information. Still, we note that the initial function, its circuit representation, and the final output of a given computation are made public during protocol execution, which in the setting of MPC is not problematic since the exposure of this information is meant to occur.

Additionally, MPCServe ensures **Guaranteed Output Termination** by improving a version of the BGW protocol [46], which by itself offers termination guarantees but lacks the robustness required for use in asynchronous environments with a malicious adversary. We extend this methodology and in the following chapters explore new mechanisms to achieve secure MPC despite Byzantine behavior assuming $2t + 1$ servers remain honest and operational. In addition, the Safety (the system emulates a centralized service), Liveness (all correct client requests are executed), and Secrecy (no private information is obtained if the threshold is respected) properties of confidential BFT SMR are directly inherited from the COBRA library.

Chapter 5

Protocols for Secure Computation in Semi-Honest Security

We now describe the core protocol (and subsequent algorithms) of our partially-synchronous, dynamic Multi-Party Computation (MPC) scheme. To simplify and decouple the presentation of our MPC framework, we first introduce mechanisms to ensure security as if we were in a semi-honest setting, and then, in the following chapter, extend the protocol to a fully malicious security setting.

5.1 Gate Evaluation in Semi-Honest Security

Following the aforementioned, computations in MPCServe start after a client has requested a $Compute$ operation indicating a function f to be evaluated by the servers. The function is then transformed into a circuit representation $C = \{g_1, ..., g_x\}$ composed of a series of addition and multiplication gates. Each server receives a copy of this circuit and then proceeds to evaluate f by processing each gate g_i of C step-by-step. We now present the nature of each gate that may compose the circuit to indicate how the servers behave, i.e. whether servers perform local computations or require additional communication.

We note that inputs processed by gates are either shares of client secrets (or output shares generated from previous gates) or constant values, which are defined within the finite field used by COBRA's secret sharing scheme.

5.1.1 Addition and Operation-by-Constant Gates

Recall that MPCServe's model relies on COBRA's DPSS, a *library* based on Shamir's additively homomorphic secret-sharing scheme. Assuming this setup, we can follow the standard approach of previous work [11, 46] which exploits the fact that adding two polynomial secrets s and s' is equivalent to adding shares of their respective polynomials $p_s(x)$ and $p_{s'}(x)$, resulting in a new polynomial $p_{s+s'}(x)$ with $s+s'$ as the constant term and with degree to the highest degree between $p_s(x)$ and $p_{s'}(x)$.

Addition Gates. When resolving an Addition Gate, which operates over two input shares a

and b, its output can be determined without any external communication as each server independently adds the shares directly to obtain a valid output share. Under our MPC model, each server s_i may hold shares $a = f(i)$ and $b = g(i)$, which were *correctly* generated and distributed using Shamir's Secret Sharing over degree-t polynomials $f(x)$ and $g(x)$ that bind secrets s_1 and s_2, respectively. These servers can then compute the output of the addition gate by locally adding their input shares over Shamir's finite field F, producing a correct degree-t share $h(i) = f(i) + g(i)$ of polynomial $h(x) = f(x) + g(x)$ which now hides the secret $s_1 + s_2$.

Operation-by-Constant Gates. We use the previous technique for Operation-by-Constant gates (i.e. Addition-by-Constant and Multiplication-by-Constant Gates), which take share a from polynomial $f(x)$ and constant value c as inputs and locally output a valid output share of $f(x) + c$ or $f(x) * c$, respectively. More formally, each server $s_i \in \Sigma$ may hold c and share $a = f(i)$, where c is a constant value $\in F$ and $a = f(i)$ is a share of degree-t polynomial $f(x)$ binded to secret s_1. For *addition-by-constant gates*, servers compute the output by locally adding its inputs, resulting in a degree-t share $h(i) = c + f(i)$ of the polynomial $h(x) = c + f(x)$, which hides the new secret $c + s_1$. Similarly, for *multiplication-by-constant gates*, servers obtain the output by locally multiplying its inputs, yielding a degree-t share $h(i) = c * f(i)$ of polynomial $h(x) = c * f(x)$, which hides the secret $c * s_1$.

The evaluation of addition and operation-by-constant gates is thus made by directly operating over shares (and respective commitments) stored by MPCServe's servers, enabling the computation of such gates without additional communication.

5.1.2 Multiplication Gates

In this section, we reason why the previous methodology for computing gates is insufficient to evaluate a *Multiplication Gate* in our work and present mechanisms to address such issues under our dynamic MPC setting.

Multiplication Challenges. A naive approach to evaluate a *multiplication gate* would be to follow the same approach in addition and operation-by-constant gates and locally operate over the input shares of the gate. In this scenario, each server s_i would hold shares $a = f(i)$ and $b = g(i)$ from degree-t polynomials $f(x)$ and $g(x)$, representing secrets s_1 and s_2, respectively, and the output of a *multiplication gate* would be resolved by directly multiplying the input shares to produce a correct degree-t share $h(i) = f(i) * g(i)$ of polynomial $h(x) = f(x) * g(x)$ which would hide $s_1 * s_2$. Despite the resulting polynomial $h(x)$ representing the desired value $s_1 * s_2$ as its constant term and each server holding a valid share of this polynomial, the previous approach raises problems regarding share reconstruction and computational security:

- The first issue is the fact that the resulting polynomial $h(x)$ will have degree $2t$, a value *twice as high* as the degree of its factors $f(x)$ and $g(x)$. This is critical as reconstructing the coefficients of a polynomial of degree $2(t-1)$ through interpolation requires at least $2t$ points/shares, breaking the secret sharing threshold t initially defined.

- Also, the coefficients of this newly generated polynomial $h(x)$ *cease to be random* as it can be expressed via the product of $f(x)$ and $g(x)$, allowing an adversary to guess them with non-negligible probability and reconstruct the underlying secret.

This challenge of evaluating multiplication gates is what defines the overall complexity of a multiparty computation protocol, as it requires auxiliary procedures to ensure that the resulting polynomial $h(x)$ remains of the correct degree and retains its randomness.

Semi-Honest Multiplications. To do multiplications, we follow Gennaro et al.'s approach [46] to achieve efficient and simplified BGW-style multiplications under passive adversaries. The protocol assumes an environment with $2t+1$ servers where each server $s_i \in \Sigma$ locally multiplies their shares $a = f(i)$ and $b = g(i)$ of polynomials $f(x)$ and $g(x)$ of degree t, respectively, obtaining the product share $h(i) = f(i)g(i)$. Then, each server s_i runs a *sub-share generation step* where it creates a random and uniformly distributed polynomial $z_i(x)$ with constant coefficient $z_i(0) = h(i)$ and sends the *sub-share* $z_i(j)$ to each server s_j for $1 \leq j \leq 2t+1$. Finally, each server s_j can generate a share of ab by computing the following equation over received *sub-shares*:

$$Z(j) = \sum_{i=1}^{2t+1} \lambda_i z_i(j)$$

where each λ_i corresponds to a value from the first row of an inverse Vandermonde [62] matrix of nature $(2t+1)$ by $(2t+1)$. This procedure is proven to be random as each λ_i is non-zero and exactly $n-t$ polynomials $z_i(x)$ are guaranteed to be chosen by correct servers, preserving the degree of the original polynomials.

The previous building block directly replaces the degree reduction and randomization step of the traditional BGW [11] protocol. Specifically, the linear equation $Z(j) = \sum_{i=1}^{2t+1} \lambda_i z_i(j)$ produces a degree-t share associated to a polynomial $Z(x)$ that (a) has the correct degree t, since summing degree-t polynomial evaluations does not increase the polynomial's degree, and (b) is constructed from degree-t polynomials $z_i(x)$, whose coefficients are randomly chosen, while ensuring the constant term satisfies the multiplication result $Z(0) = ab$.

5.1.3 Multiplication Limitations

We now look at the drawbacks of using the previous method for multiplying shares within the dynamic MPC setting and highlight how such weaknesses lead to practical challenges that must be addressed to establish a feasible practical MPC framework.

Vandermonde coefficients. A major issue emerges when adopting an approach based on Vandermonde coefficients λ_i to enable an algebraic simplification during multi-party computational multiplications. In particular, for each server s_j to compute a correct degree-t multiplication share $Z(j)$, it requires $2t+1$ coefficients to be able to multiply the corresponding sub-shares received from $2t+1$ different servers s_i (with each coefficient λ_i multiplying the sub-share sent from its associated server s_i). Note that these coefficients rely on the composition of the Vandermonde matrix which is constructed over the public (and unique) identifiers of each server participating

in the computation. However, our MPC model assumes an underlying BFT SMR environment where at least $3t + 1$ servers are required to tolerate t Byzantine faults. As such, the first row of the inverse Vandermonde matrix $V_{2t+1}^{-1}[0]$ (which is where the λ_i coefficients originate from) fails short in providing a sufficient number of coefficients to all participating servers as it only holds coefficients for at most $2t + 1$ servers.

Vandermonde structure. Similarly, a limitation exists linked to the Vandermonde matrix's structure. When configured as a $(2t + 1)$ by $(3t + 1)$ matrix as an attempt to enable the availability of $3t + 1$ Vandermonde coefficients λ_i for each participating MPC server, the Vandermonde Matrix ceases to be square and therefore non-invertible. This lack of invertibility is critical for fault-tolerant MPC as if we remove the ability to derive the first row of this matrix, the coefficients required to compute valid multiplication shares become impossible to obtain, causing the computation to be interrupted. Attempting to rectify this by constructing a square Vandermonde matrix of size $3t + 1$ by $3t + 1$ only aggravates the issue, as it introduces a contradiction in the context of BFT SMR systems. In the standard semi-honest multiplication protocol, we acknowledge that $2t + 1$ servers are sufficient for performing multiplications *without faults*. However, defining a Vandermonde matrix of nature $3t + 1$ consequently heightens the number of coefficients needed to $3t + 1$, which forces the usage of $3t + 1$ sub-shares from each server. This requirement sabotages the system's fault tolerance because as we operate over COBRA designed to tolerate up to t Byzantine faults assuming $3t + 1$ servers, referring that $3t + 1$ servers generate sub-shares during multiplications directly contradicts this premise.

Polynomial sub-shares. Another problem arises from the nature of polynomial sub-shares during multiplications. The semi-honest multiplication algorithm relies on the certitude that the $2t + 1$ servers generate and distribute $2t + 1$ sub-shares amongst each other. This step, however, assumes that each server receives one sub-share from each other, meaning that every server should hold $2t + 1$ sub-shares generated by disparate servers to generate multiplication shares and allow subsequent computations. That said, if any of these sub-shares fails to reach a specific server (be it due to a server crash or a network delay), that server cannot accurately compute its share $Z(i)$ as the algebraic simplification is mathematically dependent on the contributions from all $2t + 1$ sub-shares. The reason why $2t + 1$ sub-shares are essential lies in the multiplication process itself. In standard Shamir's secret sharing, a single polynomial of degree t encodes a given secret, allowing reconstruction with $t+1$ shares. However, during multiplications, the scenario shifts from a single polynomial to $2t + 1$ polynomials generated separately and locally by $2t + 1$ distinct servers. This change, combined with the dependency on the Vandermonde Matrix, forces the availability of all $2t + 1$ sub-shares to perform the required linear equations and construct multiplication shares of the correct degree.

Additionally, as our BFT SMR MPC framework assumes at least $3t+1$ servers are expected to participate in computations, a discrepancy occurs as only $2t+1$ servers are tasked with generating and distributing sub-shares to all participating servers. This elevates a vulnerability as selecting $2t+1$ servers from the $3t+1$ total leads to a situation where if one of the chosen servers becomes

corrupted, the sub-shares from the faulty server may not be properly delivered or be maliciously altered. Even if the corrupted server is replaced by another that takes over the task of generating and distributing sub-shares during the multiplication process, the new server will compute its multiplication share based on its public identifier, meaning that if other servers have already used the corrupted server's identifier to compute their multiplication share, a mismatch in the set of Vandermonde coefficients used between servers will occur. Consequently, this leads to an inconsistency in the share construction step as servers will compute multiplication shares that are inconsistent with each other, prompting a complete restart of the operation. As such, ACS primitives must be defined to address the issue of servers agreeing on the set of $2t + 1$ sub-share generator servers for a given multiplication.

Discussion. Given these challenges, there exists the need for a re-evaluation of the multiplication process within the dynamic MPC setting. Specifically, new approaches must be developed and efficient ACS primitives must be designed to ensure the reliable generation and distribution of polynomial sub-shares while maintaining the necessary fault tolerance under an asynchronous, Byzantine environment.

5.2 Practical Multiplications

In this section, we explore how a multi-party computation protocol can be enhanced to run over a practical environment that assumes the existence of asynchronous networks and faults of crashing or network delays. More concretely, we show why the semi-honest multiplication procedure requires additional mechanisms to run over asynchronous settings while also being able to guarantee output.

5.2.1 Fairness

In this section, we define the concept of semi-honest multiplications with Fairness and introduce a novel feature that helps address previously discussed computational limitations, particularly concerning servers responsible for the *sub-share generation step* during the multiplication procedure.

Sub-share Reconfiguration. Many of the current restrictions that limit multiplications based on BGW [11] [46] from being directly deployed in practical settings come from their dependency on the Vandermonde matrix coefficients and respective structure. One key obstacle comes from the standard assumption that $2t + 1$ servers generate sub-shares for one another, meaning only this subset participates in the computation. However, as we account for providing a dynamic MPC framework that allows servers to join and support computations, it is usual for more than $2t + 1$ servers to be available for the process. Consequently, to align with the BFT SMR model, the multiplication process of MPCServe follows a sub-variant where sub-shares are sent to every server s_j for $1 \leq j \leq n$, and not only a subset of $2t+1$, so that all servers can obtain the necessary data to compute multiplication shares. As such, each of the $2t + 1$ servers s_i that generate and distribute sub-shares create a random polynomial $z_i(x)$ with constant coefficient $z_i(0) = h(i)$ and

send the sub-share $z_i(j)$ to each server $s_j \in \Sigma$ that is participating in the computation, allowing the size of the Vandermonde matrix V_{2t+1} to remain unchanged.

Additionally, the multiplication protocol assumes a setting where the λ_i coefficients are multiplied with the initial product shares $h(i) = f(i)g(i)$ during the sub-share generation step to simplify computations for the receiving servers. By pre-multiplying the λ_i coefficients, servers generating sub-shares can directly multiply their product share $h(i)$ using the λ_i associated with their public identifier. This allows the receiving servers to focus solely on tracking the set of $2t+1$ servers responsible for generating/delivering sub-shares for a given multiplication and then performing the sum of received sub-shares to compute their final multiplication share (without the additional step of multiplying each sub-share by its corresponding coefficient).

Multiplication with Fairness. For servers to agree on the set of $2t + 1$ servers that generate and distribute sub-shares for each multiplication gate of a given circuit, we define our ACS primitive by introducing the concept of *Fairness*, which leverages the ability to *dynamically* reconfigure the Vandermonde matrix V_{2t+1} for each multiplication. We take advantage of the fact that coefficients λ_i from the first row of the inverse Vandermonde matrix $V^{-\vec{1}}_{2t+1}[0]$ depend on the public identifiers i of the set of servers s_i involved in the *sub-share generation step*, and define an environment where each multiplication deterministically defines a subset of servers (and respective set of $2t + 1$ coefficients) that participate in generating sub-shares. Here, one multiplication could rely on servers s_1, s_2, s_3 to generate sub-shares, defining the Vandermonde matrix over the public identifiers $(1, 2, 3)$, while a future multiplication could depend on a different set of servers s_1, s_2, s_4, defining the Vandermonde matrix based on identifiers $(1, 2, 4)$, and so on.

We recall that to support an ACS primitive in asynchronous MPC, an obvious approach would be to use a Byzantine agreement sub-protocol, which in our case could be directly invoked from the BFT SMR layer for servers to propose a new set of $2t + 1$ servers to perform the *sub-share generation step* for each multiplication. However, instead of using consensus for every multiplication gate, which significantly hinders computational efficiency, MPCServe implements *Fairness* through an $\underline{mult_{servers}}$ attribute, an array maintained by each server s_i that tracks the set of $2t+1$ servers participating in the *sub-share generation step* of the multiplication algorithm. Also, to maintain server consistency, $mult_{servers}$ is updated through the $\underline{mult_{count}}$ counter, a parameter held by each server s_i that tracks the number of successfully computed multiplication gates for that server. As such, our described environment works by (a) predefining a list containing all possible combinations of $2t + 1$ from $3t + 1$ servers, which is deterministic and known to all servers at the start of computation and (b) allowing servers to select the set of $2t+1$ servers by traversing this list while accounting for the current multiplication counter (and currently known delayed/malicious servers, as we later explain). This arrangement ensures that each server has up-to-date information about the server set, allowing them to construct a Vandermonde matrix based on the identifiers of these $2t + 1$ servers and use the appropriate λ_i coefficients when computing multiplication shares.

We say that the BGW-style multiplication algorithm provides *Fairness* if the set of participants involved in computation rotates for each independent multiplication gate while ensuring that at

least $2t + 1$ participants can acquire a valid output share of multiplication. This optimization allows rotating execution responsibilities, preventing the same $2t + 1$ servers from handling the entire multiplication algorithm and reducing the risk of adversaries corrupting the same set of servers in scenarios where $n > 2t + 1$ servers coexist.

5.2.2 Sub-Share Forwarding

In this section, we define multiplications with sub-share forwarding to address the computational limitation regarding sub-share dependency and respective availability.

Context. We acknowledge that sub-shares generated by the $2t + 1$ servers that perform the *sub-share generation step* are essential for producing multiplication shares, meaning that if any server fails to deliver its sub-share, the multiplication process ceases to end. Still, there are ways to mitigate this issue within our MPC environment. A direct approach to provide sub-share recovery is through COBRA's DPSS recovery protocol, enabling servers to rebuild their state - and respective sub-shares - with the aid of $t + 1$ honest servers. However, relying on consensus for each lost sub-share again introduces performance bottlenecks that can otherwise be avoided. For this matter, one effective strategy comes by *securely* storing sub-shares received from other servers, aiding scenarios where a server becomes slow or fails to deliver its sub-share, other servers that may have previously received it can forward the missing sub-share, ensuring all participants can compute valid multiplication shares without additional delay.

Proposals. As such, we implement a proposal ($proposal_{z(x)}$) to enable each server participating in the *sub-share generation step* to organize and aggregate sub-shares generated from its local polynomial $z(x)$. Moreover, each position within a $proposal_{z(x)}$ corresponds to a specific sub-share intended for a particular server in the computation set Σ. As such, if we assume n to be the number of servers in $|\Sigma|$, a proposal is defined as:

$$proposal_{z_i(x)} = \langle z_i(1), z_i(2), ..., z_i(n) \rangle$$

where $z_i(j)$ for $1 \leq i \leq n$ is the sub-share generated by server s_i during *sub-share generation step* to be sent to server s_j. This way, we rearrange the *sub-share generation step* of the semi-honest multiplication procedure and allow for server s_i to send $proposal_{z_i(x)}$ containing encrypted sub-shares associated to $z_i(x)$ instead of only the sub-share $z_i(j)$. This adjustment is relevant for multiplications as it enables each server to retain proposals $proposal_{z_j(x)}$ received from other servers s_j and forward them if needed. To ensure that each server can only access its designated share, each sub-share $z_i(j)$ of a $proposal_{z_i(x)}$ is encrypted using the respective public key $PubKey_j$ of server s_j, which is known by every server.

These measures allow servers to hold and forward encrypted sub-shares on behalf of other servers without compromising security, aiding scenarios where a server is waiting for a specific sub-share but the original server responsible for generating it is delayed or has crashed. As such, servers holding the proposal $proposal_{z(x)}$ that contains the required sub-share can step in and send it directly to the requesting server, significantly boosting multiplication performance. We

now present the updated semi-honest multiplication procedure satisfying *Fairness* and using the *proposals* just described.

Protocol 1: BGW-Style Semi-Honest Multiplication (w/ Fairness & $Proposals$)

-**Requires:** Input of each server $s_i \in \Sigma$: shares $a = f(i)$ and $b = g(i)$.

1. Each server s_i that belongs to the set of $2t + 1$ servers defined by $mult_{servers}$ shares $\lambda_i f(i)g(i)$ (where λ_i is the Vandermonde coefficient associated to server s_i) by choosing at random a degree-t polynomial $z_i(x)$ such that $z_i(0) = \lambda_i ab$. s_i then generates sub-shares $z_i(j)$ of polynomial $z_i(x)$ for each server $s_j \in \Sigma$ and defines the sub-share's proposal $proposal_{z_i(x)}$, containing each generated sub-share encrypted with the public key $PubKey_j$ of s_j.

2. Each server $s_j \in \Sigma$ computes its multiplication share over the obtained $2t+1$ sub-shares (decrypted using its private key $PriKey_j$ over the received proposals) sent by servers s_i that belong to the set of $2t + 1$ servers defined by $mult_{servers}$. s_j then computes its multiplication share $Z(j)$ of the degree-t random polynomial $Z(x)$ using $Z(j) = \sum_{i \in mult_{servers}}^{2t+1} z_i(j)$.

5.3 Tolerating Faults

In this section, we define mechanisms to handle server crashes and message delays to complement the *Fairness* and *Share Recovery* mechanisms to enable multiplications with guaranteed output assuming the presence of passive server faults.

Context. It is well known that in partially-synchronous environments it is impossible to distinguish a corrupt server that does not send any messages from an honest server whose messages have been delayed [31]. Inevitably, and despite *fairness* and *sub-share forwarding*, mechanisms must exist to account for scenarios where a sub-share becomes entirely unrecoverable — such as when an adversary correctly guesses the set of servers that generate sub-shares and blocks a server from generating/delivering any of its sub-shares. Hence, a fallback strategy is necessary for the system to roll back and ensure that no server is left in an inconsistent state due to missing data.

Fault-Tolerant Multiplications. To provide fault tolerance, we propose a new algorithm $Mult_{Delay}$(Algorithm 5.3) to handle scenarios where a server does not send the required messages/shares (due to network delays or crash) within a predefined time interval $timeout$ for a given multiplication defined by $mult_{count}$. We rewrite $proposal_{z_i(x)}$ as $proposal_i$ for simplicity purposes.

Protocol 2: $Mult_{Delay}$ - Delayed Servers Procedure for multiplication $mult_{count}$

-**Assumes:** Each server s_i that belongs to the set of $2t+1$ servers defined by $mult_{servers}$ has attempted to generate and broadcast $M_{proposal_i} : Sign_i(proposal_i)$ to each server $\in \Sigma$, where $proposal_i = \langle PubKey_1(share_1), \ldots, PubKey_n(share_n)\rangle$ ($n = |\Sigma|$).

1. Each server $s_i \in S$ starts timeout t_1.
   1.1. If timeout t_1 ends before server s_i receives $2t+1$ proposals needed for the multiplication, then s_i starts timeout t_2, defines $server_{IDs}$ (which are IDs of servers from which no proposals were received), and broadcasts $M_{request\_proposal} = (mult_{count}, server_{IDs})$ to servers $\in S$.
   1.1.1. If timeout t_2 ends before receiving the remaining needed proposals for multiplication, then s_i broadcasts $M_{delayed\_server} = (mult_{count}, server_{IDs})$ to servers $\in S$.
2. Upon receiving $M_{request\_proposal}$ from s_j, each server $s_i \in S \setminus \{s_j\}$:
   2.1. Checks if it has the requested proposal for server s_j, and if so, it sends message $M_{proposal_i}$ containing the proposal back to server s_j.
3. Upon receiving $t+1$ $M_{delayed\_server}$ from different s_j, each server $s_i \in S \setminus \{s_j\}$:
   3.1. Adds $server_{IDs}$ to its $delayed_{servers}$.
4. Upon receiving $M_{proposal_j}$ from s_k, each server $s_i \in S \setminus \{s_j\}$:
   4.1. Checks validity of received proposal and, if it is valid, stores $proposal_j$ from message ($M_{proposal_j}$ and removes s_k from $server_{IDs}$.

The $Mult_{Delay}$ algorithm starts by each server $s_i \in \Sigma$ initiating a timeout t_1 during which they wait to receive the required sub-shares for that multiplication. If this first timeout ends for server s_i without receiving the $2t+1$ necessary proposals, that server starts a new timeout t_2 and broadcasts a complaint as messages $M_{request\_proposal}$ requesting the missing $proposals$ containing the sub-shares needed to complete its multiplication gate. During this last process, if the second timeout t_2 that was initiated ends for server s_i without receiving all necessary proposals to finalize its multiplication, that server broadcasts an accusation against servers that failed to submit their proposals in timeouts t_1 and t_2.

A message $M_{proposal_j}$ refers to a message containing $proposal_j$, the proposal generated from server s_j during the given multiplication. Nevertheless, as proposals can be forwarded by other servers that have previously received them, any server possessing $proposal_j$ can facilitate the request by forwarding the proposal on behalf of the original generating server s_j.

A message $M_{request\_proposal}$ is composed of the current multiplication counter $mult_{count}$ of the sender and the identifiers of servers that did not respond with the required sub-shares ($server_{IDs}$) during timeouts. Upon receiving $M_{request\_proposal}$, servers compare their current multiplication counter with the one included in the request and check if they possess the requested proposals associated with the servers listed in the received $server_{IDs}$ by verifying whether they have previously received proposals from the specified servers. If such proposals are available, the server holding the necessary proposals forwards them to the requesting server.

Similarly, a message $M_{delayed\_server}$ is composed of the current $mult_{count}$ from the sender and the identifiers of servers that did not respond with the required sub-shares ($server_{IDs}$). As any subset of $t + 1$ servers assumes at least one honest server, a server is considered as $delayed$ if it receives $t + 1$ complaints from different servers for a given multiplication $mult_{count}$. To temporarily remove a $delayed$ server from a multiplication, the implementation of MPCServe defines the $delayed_{servers}$ attribute, an array maintained by each server $s_i \in \Sigma$ of size t that records the most *recent* set of servers flagged as $delayed$. As such, when a server detects that another possibly faulty server has not sent its sub-shares *in time*, the $Mult_{Delay}$ algorithm allows rolling back execution, remove old sub-shares shared during the corrupted multiplication, and update the multiplication set by selecting a new group of $2t + 1$ servers, which is found by performing an intersection between $mult_{servers}$ and $delayed_{servers}$. This way, a new multiplication procedure can be initiated assuming the revised $mult_{servers}$ which factors in the newly identified delayed servers.

5.4 Security

We informally analyze the security of the presented protocols related to multiplications of multi-party computations under a passive adversary capable of (a) observing external behavior of the system (b) learn the state of t passively corrupted servers or (c) crash or delay up to t servers.

Fairness. The dynamic rotation of the $2t + 1$ servers involved in generating sub-shares during multiplications provides security in the context of semi-honest multi-party computations. If we assume that the set of servers remained static, an adversary could target one of these known servers and attempt to disrupt or delay the multiplication process. However, by implementing Fairness where the set changes for different multiplications, the adversary faces a reduced probability of correctly guessing which servers are delegated to generate sub-shares for a given multiplication. On the other hand, an internal adversary that passively observes the server's state may be capable of identifying the $2t + 1$ servers generating sub-shares in a specific multiplication if it targets a previously correct server. Nevertheless, the remaining mechanisms - Sub-share Forwarding and Fault-Tolerant Multiplications - work together to ensure that any detected abnormal behavior triggers a reconfiguration, resulting in the set of servers being switched to a new group and preventing passively corrupted servers from guessing sets for subsequent multiplications.

Sub-share Forwarding. From a security perspective, sub-share forwarding helps address potential failures or delays without compromising security by mitigating two threats: passive eavesdropping and server failure. Each sub-share $z_i(j)$ within a proposal $proposal_{z_i(x)}$ is encrypted with the recipient server's public key $PubKey_j$, meaning that even if a sub-share is forwarded by an intermediate server, only the intended recipient can decrypt and access the data. As such, a passive adversary that compromises a server by gaining access to its stored proposals would not be able to extract sub-shares intended for other servers, as they would not possess the required keys to decrypt them. Furthermore, by using proposals for servers to store and forward encrypted sub-shares when the original server is unavailable, the mechanism helps avoid performance bottle-

necks that would arise from having to initiate a full roll-back every time a sub-share (and respective proposal) becomes delayed.

Fault-Tolerant Multiplications. The $Mult_{Delay}$ algorithm uses a timeout-based approach with sub-share forwarding and accusation threshold. More concretely, timeouts allow for server delays to be addressed via proposal forwarding from honest servers, meaning that even if t servers fail to deliver their proposal in the first timeout, other servers can step in and provide the required proposal without needing to roll back the multiplication. In case no proposals are available, the $Mult_{Delay}$ relies on a specific threshold of accusations where at least one honest server is involved during the complaint process.

This way, the algorithm defines a condition where if any server detects a significant delay in receiving the necessary proposal, $t + 1$ accusations from different servers are sufficient and necessary to define a delayed server within the MPC setting. The algorithm ensures that each server consistently holds the same $2t + 1$ multiplication server set $mult_{servers}$ for a given multiplication, as information about the current multiplication set is propagated to all honest servers through accusations - which include at least one accusation from an honest server - in a deterministic fashion assuming the underlying authenticated fair channel network (where messages can be delayed, but not forever).

Discussion. The previous enhancements for the semi-honest multiplication protocol in Section 5, namely:

- **Fairness** to prevent the same set of $2t+1$ servers of performing sub-share generation during multiplications;
- **Sub-share Forwarding** for servers to forward missing sub-shares, averting computation delays caused by slow or non-responsive servers;
- **Fault-Tolerance** to exclude faulty servers and ensure computational continuity over an updated set of $2t + 1$ servers.

lead to partially-synchronous, dynamic multi-party computations with guaranteed output under a semi-honest security setting. Assuming that the adversary may only externally observe the system behavior and access the server state, the threshold $t \leq (n - 1)/2$ is sufficient to ensure secure computations since $2t+1$ servers will remain online to generate and share the required sub-shares during multiplications. To additionally provide security under the assumption that servers can crash and might fail to deliver their proposals to any other server, the threshold increases to $n \geq 3t + 1$ to ensure that even if up to t servers fail, at least $2t + 1$ servers can reliably generate and send proposals to each other and ensure guaranteed computational output.

Chapter 6

Protocols for Secure Computation in Malicious Security

In this chapter, we extend the enhanced semi-honest multiplication protocol of the previous chapter to achieve computational security against an active adversary corrupting up to $t < n/3$ servers. For this goal, we define mechanisms for (a) share verification (b) solving malicious behavior (i.e. servers sending invalid sub-shares during multiplications), and (c) proving share ownership.

6.1 Verifiable Secret Sharing

In this section, we present the Verifiable Secret Sharing (VSS) approach wielded by MPCServe to ensure share validation during computations.

Context. Despite improvements introduced to ensure semi-honest multiplications supposing server crashes, namely Fairness, Sub-share Forwarding, and Fault Tolerance, practical MPC must additionally be resilient to a stronger, more realistic adversary that can actively compromise computations by changing the protocol's execution. Specifically, a malicious server may attempt to generate and distribute invalid sub-shares during the *sub-share generation step* of multiplications which, if undetected, can temper computational integrity by corrupting the production of degree-t multiplication shares of other servers. From that point onward, all subsequent gates that depend on the output of that multiplication will operate over incorrect values, compromising the entire computation.

Verifiable Shares. To ensure servers identify invalid sub-shares and operate over correct values during multiplications, MPCServe leverages COBRA's DPSS implementation of Feldman's [45] additively homomorphic commitments to grant *share verification* during computations. We chose Feldman's scheme instead of other popular commitment schemes such as Pedersen's [45] due to its simplicity, as it is based on coefficients rather than points/shares of polynomials; and efficiency, as it avoids computational overhead by operating over a single polynomial instead of an additional polynomial used for randomness. This line of reasoning defines primitives that enhance multiplication security by validating share a_i of polynomial $f(x)$ using its respective

commitment $C_{f(x)}$ without requiring knowledge of the underlying secret $f(0) = s$.

We extend the *sub-share generation step* of the semi-honest multiplication in Section 5.2.2 and change its implementation to enable any server to generate verifiable sub-shares vs_{a_i} (instead of traditional sub-shares a_i), encapsulate them into $proposals$, and broadcast *verifiable* $proposals$ to servers $s_i \in \Sigma$. This way, we define an environment where each server s_j can receive a *verifiable* $proposal$ from any other server s_i and check the validity of its intended sub-share $a_j = f(j)$ by verifying:

$$g^{f(j)} = \prod_{k=0}^{t} C_k^{j^k}$$

Consequently, any server s_j can correctly identify if a malicious server s_i has generated and delivered an invalid share a_j by checking if the previous equation holds.

6.2 Dealing with Malicious Servers

In this section, we define mechanisms to address maliciously identified servers during multiplications and achieve computational continuity under a malicious security setting.

Context. We established that resolving a malicious adversary who corrupts up to $t < n/3$ servers during computations requires accurate identification of servers that deviate from the intended protocol. Consequently, mechanisms must be implemented to address malicious servers (and corresponding tampering) identified during multiplications and ensure that computational integrity remains intact.

Managing Invalid Shares. Extending the previously defined verification scheme (Section 6.1), we present $Mult_{Accuse}$, an algorithm that handles scenarios where a Byzantine server sends invalid sub-shares during a given multiplication $mult_{count}$.

Protocol 3: $Mult_{Accuse}$ - Invalid Share Procedure for multiplication $mult_{count}$

-**Assumes:** Each server s_i that belongs to the set of $2t+1$ servers defined by $mult_{servers}$ has attempted to generate and broadcast $M_{proposal_i} : Sign_i(proposal_i)$ to each server $\in \Sigma$, where $proposal_i = \langle PubKey_1(share_1), \ldots, PubKey_n(share_n)\rangle$ $(n = |\Sigma|)$.

1. Upon receiving $M_{proposal_k}$ from s_j, each server $s_i \in S \setminus \{s_j\}$:
   1. Checks if both the signature of the received message $M_{proposal_k}$ and the signature of the proposal within $M_{proposal_k}$ are valid.
   1.1. If yes, check if the sub-share $share_i$ (decrypted using its private key $PriKey_i$ over its respective encrypted sub-share of the received proposal $proposal_k$) is invalid.
   1.1.1. If yes, it adds server s_k (who generated the malicious sub-share) to its malicious list $malicious_{servers}$ and broadcasts *accusation* $M_{accuse\_proposal_k} = Sign_i(k, M_{proposal_k}, share_i)$ to every server $\in \Sigma$.

2. Upon receiving $M_{accuse\_proposal_k}$ from s_j, each server $s_i \in S \setminus \{s_j\}$:
   2. Checks if both the signature of the received message $M_{accuse\_proposal_k}$ and the signature of the proposal $M_{proposal_k}$ within $M_{accuse\_proposal_k}$ are valid.
   2.1. If yes, checks if the sub-share $share_j$ received in $M_{accuse\_proposal_k}$ is invalid and, if encrypted with $PubKey_j$, is equivalent to the sub-share for server s_j present in $M_{proposal_k}$:
   2.1.1. If yes, it adds server s_k (who generated the malicious sub-share) to its malicious list $malicious_{servers}$.

The presented algorithm follows the same methodology used in the share recovery protocol defined in Section 5.3, where proposals exchanged between servers might not originate from the server that initially created them. Thus, the algorithm is written in such a way that $M_{proposal_k}$ was sent by server s_j, but does not guarantee that server s_j has generated $M_{proposal_k}$. In other words, $M_{proposal_k}$ is generated by server s_k but may be delivered by another server other than s_k if it already holds $M_{proposal_k}$.

The routine starts with the standard multiplication procedure, where each server that performs the *sub-share generation step* ($s_i \in mult_{servers}$) generates and broadcasts a message containing a *verifiable* $proposal$, composed of *verifiable* sub-shares, to each server s_j involved in the computation. Upon receiving a $proposal$, a server s_i checks whether the obtained multiplication $proposal$ is valid by checking the sender's message signature (and the signature of the $proposal$) and then verifying the validity of the respective $proposal$'s sub-share $share_i$ via the verifiable scheme defined in Section 6.1. If this last verification process fails, the malicious server s_k associated with generating an invalid sub-share is added to the list of malicious servers $malicious_{servers}$ of size at most t, is excluded from computation by removing it from $mult_{servers}$, and an accusation is made to the corrupted server s_k over the invalid sub-share $share_i$. When server s_i receives an accusation from a server s_j, it verifies the message signatures and compares the accused sub-share $share_j$, encrypted using the sender's public key, with the encrypted sub-share contained within the original proposal. If these two match, it confirms that the sub-share is indeed the same and invalid, establishing the server that generated the original proposal to be malicious, enabling s_i

to add the accused server to its own $malicious_{servers}$ list. We note that messages containing incorrect signatures are treated and processed similarly to invalid sub-shares.

Moreover, MPCServe's implementation complements the $Mult_{Accuse}$ algorithm by including mechanisms to dynamically identify and exclude malicious servers without disrupting ongoing computations, by first halting the compromised multiplication, then mitigating abnormal activity by excluding malicious servers, and retrying the computation over an updated set of servers. This way, we ensure that multiplications proceed correctly and prevent the propagation of invalid output shares to future gates, safeguarding the integrity of evaluated circuits.

6.3 Proving Share Ownership

In this section, we define cryptographic proofs for servers to manifest share ownership during multiplications.

Context. We acknowledge that evaluating a multiplication gate requires servers to communicate with each other to compute the product of their private shares $a = f(i)$ and $b = g(i)$. In this context, a fraudulent server could manipulate the *sub-share generation step* during a multiplication by generating sub-shares (and respective commitments) from a fake value instead of generating sub-shares from the product of its shares $ab = f(i)g(i)$, leading the remaining servers to believe that their received sub-shares are consistent and valid. As a result, servers would end up with a multiplication share generated based on incorrect sub-shares, compromising the remainder evaluation of the underlying circuit.

Share proof. As such, we introduce a Zero-Knowledge proof [49] for servers to prove that sub-shares (and respective commitments) created during multiplications were generated based on previously stored and correct shares. We take inspiration from Pedersen's commitments [69] and define an algorithm to run during multiplications which allows a server, the Prover P, to prove to another server, the Verifier V, that it can *open* $A = g^a$ and $B = g^b$. In other words, he knows shares a and b and that the opening of $C = g^{ab}$ is related to the product of the values it committed to A and B. Our proof (Section 6.3) enhances previous ZK proofs for the MPC setting [46, 27] by adopting a Non-Interactive Zero-Knowledge [16] approach which is possible through Fiat-Shamir's heuristic [13]. This way, we replace the interactive challenge-response step of the standard (and interactive) ZK proofs with a cryptographic hash function H to reduce the communication complexity from three-round interactions to a single message exchange.

Protocol 4: $Mult_{NIZK}$ - NIZK Proof for Share Multiplication

-**Requires:** Public Input: $A = g^a$, $B = g^b$, $C = g^{ab}$.

1. P chooses random $d, x \in Z_q$ and generates $M = g^d$, $M_1 = g^x$, $M_2 = B^x$.
2. P creates challenge $e = H(A, B, C, M, M_1, M_2)$, where H is a hash function.
3. P generates $y = d + eb$, $z = x + ea$ and sends $Pr = (M, M_1, M_2, e, y, z)$ to V.
4. V verifies if $g^y = MB^e$, $g^z = M_1A^e$, $B^z = M_2C^e$.

To permit proving of the correctness of shares, the presented NIZK proof is run by each server (acting as the Prover P) participating in the *sub-share generation step* of multiplications and broadcasted to each other server $s_i \in \Sigma$ (acting as the Verifier V), allowing each server to prove and verify that the generated sub-shares are derived from the correct multiplication of two valid shares.

6.4 Multiplications in Malicious Security

With the tools developed in the previous sections, this section serves to present our final dynamic multiplication procedure which allows for secure multi-party computations with guaranteed output against $t < n/3$ active adversaries under an asynchronous environment.

Secure Multiplication. We combine the enhanced semi-honest multiplication procedure in Section 5.2.2 together with our verifiable secret sharing mechanisms (Section 6.1) and NIZK proof (Section 6.3) to define secure multiplications assuming a malicious security setting, in Protocol 6.4. In addition, this newly proposed multiplication procedure is aided by previously defined algorithms that handle delayed/crashed servers (Section 5.3) and invalid shares (Section 6.2) which are executed as background tasks and triggered when necessary.

Protocol 5: *Mult* - MPCServe's Multiplication Procedure

-**Requires:** Private Input of server $s_i \in \Sigma$: $a_i = f(i), b_i = g(i)$; Public Input: $C_{f(x)}, C_{g(x)}$.

1. **Secret sharing:** Each server s_i that strictly belongs to the set *of* $2t+1$ *servers defined by* $mult_{servers}$ secret shares $\lambda_i a_i b_i$, where λ_i is the respective Vandermonde coefficients. Let $z_i(x)$ be the polynomial of degree t used in this secret sharing such that $z_i(0) = \lambda_i a_i b_i$ and let the output be the list $proposal_{z_i(x)} = (z_i(0), ..., z_i(n))$, where each position j defines the sub-share for server $s_j \in S$. Each sub-share $z_i(j)$ is also encrypted with the respective server's public key $PubKey_j$. Then, each server s_i strictly belonging to the set of $2t+1$ servers defined by $mult_{servers}$ broadcasts its proposal $proposal_i$ for every server $s_j \in S$ currently participating in the computation, together with $C_{z_i(x)}$ (the Feldman public commitment related to polynomial $z_i(x)$), and Zero-Knowledge Proof public data (equivalent to $Mult_{NIZK}$'s Pr data).

2. **Sub-Share Verification:** Each server $s_j \in S$ participating in the computation verifies each received sub-share of $proposal_i$ from servers s_i belonging to the set of $2t+1$ servers defined by $mult_{servers}$ together with the published commitment $C_{z_i(x)}$. If a received sub-share is invalid, it calls $Mult_{Accuse}$ on s_i.

3. **NIZK Proof Verification:** Each server $s_j \in S$ participating in the computation computes $A_i = g^{a_i}$ and $B_i = g^{b_i}$ of s_i using commitments C_{a_i} and C_{b_i} (which are publicly known) and the remaining received Zero Knowledge data sent from s_i. Then, s_j is ready to act as the Verifier and check if the proof sent by the Prover s_i is valid. If the received proof is not valid, s_j exposes s_i as malicious if $Mult_{NIZK}$ fails.

4. **Compute Degree-t Share:** Each server $s_j \in S$ participating in the computation computes $Z(j) = \sum_{i \in mult_{servers}}^{2t+1} z_i(j)$, which is a share of ab from polynomial $Z(x)$ of degree t, and also computes the share's commitment $C_{Z(x)} = \prod_{i \in mult_{servers}}^{2t+1} C_{z_i(x)}$.

Output: Degree-t Multiplication Share $vs_{Z(j)} = (Z(j), C_{Z(x)})$ for each server $s_j \in \Sigma$.

The protocol begins similarly to the semi-honest version: each server in the chosen set of $2t+1$ servers, defined by $mult_{servers}$, multiplies the input shares of the multiplication gate (along with the corresponding Vandermonde coefficient), generates sub-shares of a random polynomial hiding the product result, and broadcasts a proposal containing the encrypted sub-shares together with the polynomial commitment. Each server currently participating in the computational effort receives these proposals and verifies the sub-shares following the verification scheme in Section 6.1, calling the $Mult_{Accuse}$ procedure if needed. Additionally, the $2t+1$ sub-share-generating servers distribute zero-knowledge data following the $Mult_{NIZK}$ procedure to prove the correctness of the sub-share generation step and ensure that sub-shares originate from the product of two valid shares. After verifying sub-shares and validating proofs, each server computes a final multiplication share by applying the same linear equation as in the semi-honest approach, using the $2t+1$ sub-shares from different servers' proposals. Similarly to the operations that occur over sub-shares, the commitment to the final multiplication share can then be computed as $C_{Z(x)} = \prod_{i \in mult_{servers}}^{2t+1} C_{z_i(x)}$, corresponding to the equation:

$$C_{Z(x)} = \langle g^{\sum_{i \in mult_{servers}} \text{coefficient}_{i,0}}, g^{\sum_{i \in mult_{servers}} \text{coefficient}_{i,1}}, \ldots, g^{\sum_{i \in mult_{servers}} \text{coefficient}_{i,t}} \rangle$$

where $\text{coefficient}_{i,j}$ corresponds to the $j'th$ coefficient c_j of the $i'th$ polynomial (assuming $f(x) = c_{t-1}x^{t-1} + ... + c_1x + c_0$).

6.5 Security

We now informally analyze the security of MPCServe assuming a malicious adversary that can arbitrarily deviate from protocol execution. We analyze each defined mechanism that addresses malicious behavior, including the verifiable scheme for detecting invalid shares, the algorithm to manage malicious servers during multiplications, zero-knowledge proofs to ensure share correctness, and the final multiplication protocol for secure computations.

Verification Scheme. Our verification scheme is secure under a dishonest majority setting, as we assume an environment where each share maintained by MPC servers is supported by its corresponding Feldman's commitment to enable share verification possible under the discrete log hardness assumption. However, to ensure guaranteed output in our MPC setting, we shift to an honest-majority model with $n \geq 3t + 1$ total servers to tolerate up to t malicious faults and allow at least $2t + 1$ servers to complete multiplications.

Malicious Server Accusation. The security of the $Mult_{Accuse}$ algorithm is built around the use of signatures and the integrity of proposals. More concretely, the algorithm prevents false accusations by requiring that all accusations reference the original proposal, which is signed by the generating server and cannot be altered without invalidating the signature. Since the accusation directly points to a signed proposal, we emphasize that a single accusation is sufficient to identify a malicious server. Under this context, if any sub-share $share_i$ is confirmed to be invalid, it can be sent via plaintext during an accusation as it is no longer usable for the current or future multiplications.

Furthermore, if any malicious activity is detected within the set of servers that generate and distribute sub-shares, each honest server is obligated to roll back to the state before the corrupted multiplication as if invalid shares were used in computing the multiplication at $mult_{count}$, the future evaluation of subsequent gates of the circuit may be compromised. Despite this, rollbacks have minimal impact on the efficiency of multiplications as in typical scenarios of the protocol execution, a rollback will involve only a single multiplication gate. This is because for a server to complete one multiplication, it requires data from at least $2t$ other servers (in the best-case scenario), meaning that for a rollback to affect more than one multiplication gate, a server would have to detect a malicious share only after $2t + 1$ servers have already completed their multiplications over correct sub-shares.

Zero Knowledge Proofs. The presented NIZK proof for share correctness during multiplications is secure under the assumption that the chosen hash function H behaves as a Random

Oracle [8], ensuring that the challenge e is truly random and unpredictable. Additionally, the reliability of our proof is strengthened by adopting a strong methodology [12] of the Fiat-Shamir heuristic which, unlike weak proofs that only hash the commitment, we hash both the commitment and the statement to be proved. This choice ensures that even if a verifier behaves dishonestly, it cannot extract additional information from the proof beyond the correctness of the statement.

Also, the NIZK proof particularly fits our MPC setting because shares held by each server are originally generated and sent by honest clients (as per COBRA's security model), meaning these shares and respective commitments are correct. This means that when servers compute addition or multiplication gates over these initial shares, at least $2t + 1$ public commitments associated with the output shares of such operations will remain equal across $2t + 1$ honest servers, as the commitments are linked to the coefficients of the underlying polynomials. Hence, the NIZK proof allows addressing a scenario where a malicious server provides sub-shares and commitments generated from polynomials separate from the intended ones (in such a way that it bypasses the verification scheme) by ensuring that sub-shares are derived from the correct product of two legitimate shares whose commitments are known by $2t + 1$ servers. On the other hand, if a server only changes its share to be incorrect but generates a valid commitment, the share's inconsistency is revealed via our verification scheme.

Secure Multiplications. Finally, we can conclude that the $Mult$ algorithm ensures security against a malicious adversary by addressing all critical aspects that can damage the coherence of MPC computations through the previously presented mechanisms: non-synchrony is managed through COBRA's BFT SMR communication channels, which ensure reliable communication in a partially-synchronous setting; fault tolerance is ensured by temporarily excluding servers that may be slow or have crashed ($Mult_{Delay}$); malicious behavior (i.e. invalid shares) is detected via the verification scheme and NIZK proofs; and byzantine server removal is addressed by the $Mult_{Accuse}$ algorithm, allowing to roll back computations to a safe state and ensuring computations proceed with $n \geq 3t + 1$ servers assuming t maliciously corrupted servers.

Chapter 7

Evaluation

In this chapter, we present the experimental evaluation of our MPCServe multi-party computational framework. We aim to demonstrate the impact that each mechanism imposes on our system, starting with the complexity analysis, the performance of individual addition and multiplication gates, the performance of small and large circuits with different compositions, and the latency caused by a server failing or behaving maliciously.

7.1 Complexity Analysis

We now summarize the communication and round complexity analysis of the core protocols involved in MPCServe. We consider both the cost in the "best case" scenario where servers follow the protocol specification and the "worst case" scenario where some server deviates by presenting faulty or malicious behavior. We focus on communication complexity as, unlike methodologies that rely on computational overhead (i.e. Fully Homomorphic Encryption), the efficiency of multi-party computation frameworks greatly depends upon the latency imposed by communication. This line of thought is further supported by previous work, which prioritizes communication complexity as a critical factor, rather than computational overhead, in optimizing performance.

7.1.1 Communication Complexity

In this subsection, we define the communication complexity associated with our MPCServe framework which is derived from the communication complexity of the maliciously secure multiplication algorithm $Mult$. More specifically, we analyze the number of messages exchanged during inter-server communication of all underlying protocols.

The best-case complexity of MPCServe is achieved when $2t + 1$ servers send a broadcast message containing a proposal to n servers in such a way that they are received correctly and used to compute multiplication shares, resulting in $O(2t + 1) \times O(n) = O(nt)$ communication complexity. In the worst case, MPCServe achieves the best-case complexity of $O(nt)$ plus the worst-case message complexity of the underlying algorithms $Mult_{Delay}$ and $Mult_{Accuse}$, as each algorithm does not chain in a way that would cause complexity to be multiplied:

Gate Type	$t = 1$	$t = 2$	$t = 3$
Addition	0.163ms	0.163ms	0.164ms
Multiplication	70ms	116ms	185ms

Table 7.1: Micro-benchmark of the average latency (in milliseconds, 1000 iterations) to compute an addition/multiplication gate for $t = 1, 2, 3$.

- For $Mult_{Delay}$, the worst-case complexity is $O(n^2)$ in the case where each server does not have a specific proposal to be forwarded (n servers broadcast to $n - 1$ servers a message requesting a missing proposal);
- For $Mult_{Accuse}$, the worst-case complexity is $O(n^2)$ in the case where each server accuses another server by invalidating a proposal (n servers broadcast to $n - 1$ servers a message containing the maliciously identified proposal).

As such, the worst-case communication complexity of MPCServe is given by $O(nt)+O(n^2)+O(n^2) = O(n^2)$.

7.2 Experimental Description

The following experiments were executed in a cluster of 11 physical machines connected through a gigabit ethernet. All machines are Dell PowerEdge R410 servers, with 32GB of memory and two quadcore 2.27 Intel Xeon E5520 processors with hyperthreading. The machines run Ubuntu Linux 20.04.1 LTS and JRE 1.8. Each test assumes system configurations of realistic size $1 \leq t \leq 3$ for $n = 3t + 1$ ($4, 7$, and 10 servers, respectively) and for each tested t, MPCServe generates all $\binom{3t+1}{2t+1} \cdot (2t + 1)!$ Vandermonde matrices via a one-time, pre-processing procedure so that servers within the specified set of $2t + 1$ servers that participate in generating sub-shares can effectively access valid Lagrange coefficients.

We define three experiments: performance analysis of individual gates, performance analysis of circuits composed of different gates, and corruption analysis assuming a faulty or malicious server. We chose this experimental set as we focus on measuring the overhead introduced by the implemented novel features and understanding our system's capability when handling circuits under different inputs and system configurations.

7.3 Gate Evaluation

In this section, we examine the average time required for MPCServe to evaluate addition and multiplication gates, and show how the system's performance varies depending on the type of the evaluated gate. For the following tests, we previously ran COBRA's Put functionality as a means for each participating server to hold the necessary shares required when evaluating a specific gate or circuit.

Circuit	**Nº Local Operations**	**Nº Multiplications**
Sum	$n-1$	0
Average	n	0
Dot Product	$n-1$	n
Sum of Squared Differences	$n/2+n/2-1$	$n/2$
Product	0	$n-1$
Sum of Cubes	$n-1$	$2n$

Table 7.2: List of circuits selected for evaluation and their gate composition. Local Operations include Additions, Subtractions, and Operations-by-Constant.

Table 7.1 presents the average performance of evaluating a single addition and multiplication gate, per 1000 iterations, for threshold $t = 1, 2, 3$ assuming $n \geq 3t + 1$ total MPC servers. For each addition gate, latency consistently remains below $1ms$ across all threshold values, which is expected as each operation is made locally and directly over stored shares (and respective commitments) of each server, and thus made extremely quickly. In contrast, the multiplication gate exhibits much higher latency, achieving $70ms$ for $t = 1$, $116ms$ for $t = 2$, and $185ms$ for $t = 3$. This is expected as servers now have to send sub-shares to each other, check the validity of distributed sub-shares, and generate new multiplication shares based on sent sub-shares. We also observe that, as the threshold t increases, the number of required servers to maintain security proportionally increases, resulting in more computation and communication between servers. Still, we feel that the performance of multiplication gates is acceptable as each one is rapidly computed in less than $200ms$ on average.

7.4 Circuit Evaluation

In this section, we present the average time required for MPCServe to evaluate different circuits assuming different compositions of varying numbers of additions and multiplication gates to demonstrate disparities in performance between such operations and report their impact on the overall circuit evaluation time.

Table 7.2 presents a list of simple circuits (Sum, Average, Dot Product, Sum of Squared Differences, Product, Sum of Cubes) to be evaluated by MPCServe. We chose the previous circuits to illustrate the performance progression from no multiplications (where no inter-server communication is required) to several multiplications, allowing us to accurately determine the performance cost of each type of gate.

Figure 7.1 shows the latency results of the first four circuits in Table 7.2 evaluated by MPCServe considering an input size of $n = 1000$ under various system configurations ($t = 1, 2, 3$). For circuits primarily involving addition gates, such as the Sum and Average, results display a constant latency of 165 milliseconds, highlighting the light computational effort needed from in-

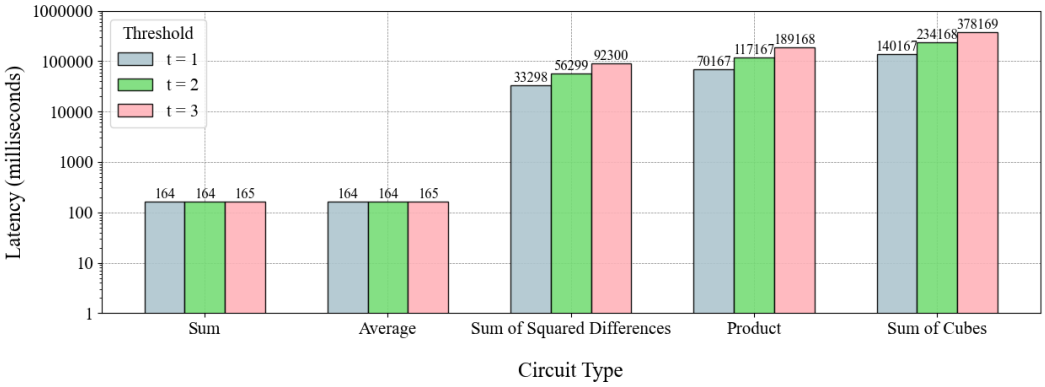

Figure 7.1: Performance evaluation of circuits in Table 7.2 from MPCServe. Y-axis (Latency) represented in logarithmic scale for readability purposes.

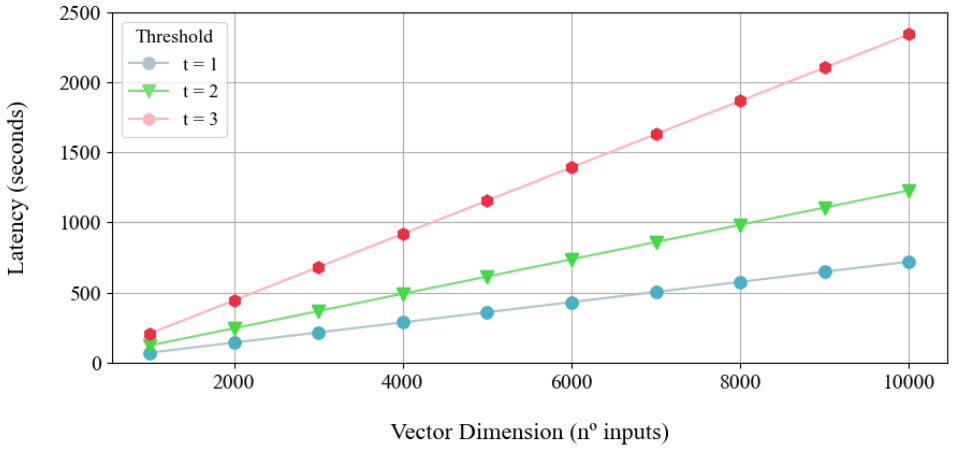

Figure 7.2: Average latency (seconds) to compute the inner product of two vectors for $t = 1$ (blue), 2 (orange), 3 (green).

dividual servers when evaluating local gates. It is possible to observe how the latency for the Sum of Squared Differences grows markedly, reaching 33 s (seconds) for $t = 1$, 56 s for $t = 2$, and 92 s for $t = 3$. In contrast, the Product circuit demonstrates latencies of 70 s for $t = 1$, 117 s for $t = 2$, and 189 s for $t = 3$, reflecting an average increase of approximately 200%, which is consistent with the evaluation of twice the number of multiplication gates. This trend continues with the Sum of Cubes, which manifests an average latency of 140 s, 234 s, and 378 s for $t = 1, 2,$ and 3, respectively. Still, and despite the time needed to complete the evaluation of such circuits, the average performance time required to compute each gate does not deviate from the standalone evaluation time exhibited in Table 7.1, with variations only differing a totality of $1-3\%$ depending on the system configuration.

Figure 7.2 further shows results of evaluating the dot product circuit, now stressed up to vectors with dimensions from 1000 to 10000, and assuming thresholds $t = 1, 2, 3$. At smaller dimensions

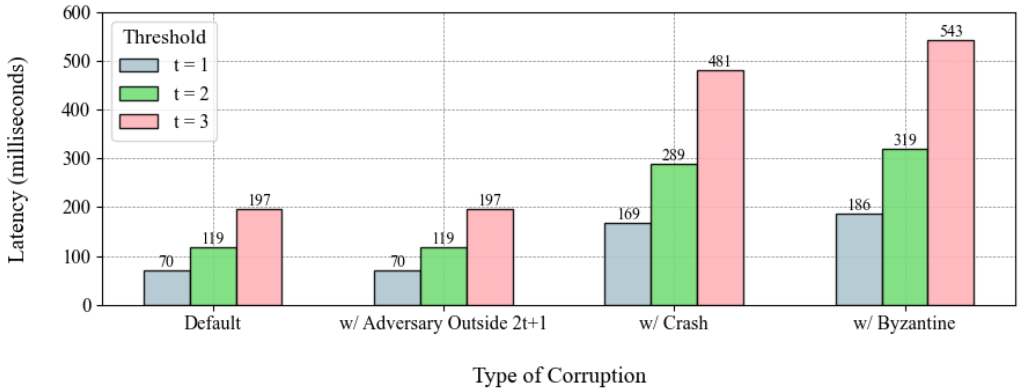

Figure 7.3: Latency (seconds) to correctly compute the default *Mult* protocol, *Mult* with an adversary outside the $2t+1$ sub-share generator server set, *Mult* with a crash, and *Mult* with invalid sub-share generation/distribution, respectively, for $t = 1$ (blue), 2 (orange), 3 (green).

($dimension \leq 1000$), the system performs well as it evaluates multiplication gates at a similar speed to standalone gates (an average increase of only $\sim 1.006\%$). As dimensions grow, the performance impact also significantly increases. For instance, at $dimension = 2000$ the latency increases $\sim 68\%$ from $t = 1$ to $t = 2$ and $\sim 80\%$ for $t = 2$ to $t = 3$, while at $dimension = 8000$ the latency from $t = 1$ to $t = 2$ remains similar ($\sim 69\%$) whereas from $t = 2$ to $t = 3$ we see an increase of $\sim 92\%$, a 12% higher rate than previously.

These circuit evaluation observations help highlight the complexity and communication overhead needed at larger input sizes and system configurations, where operations for secret-sharing, message exchange, and share/proof verification processes are fundamentally necessary.

7.5 Corruption Analysis

In this section, we observe how corrupting multiplications impact the overall performance of the MCPServe framework.

We test four core scenarios that we feel are relevant in illustrating the dependencies to each background mechanism of the multiplication protocol *Mult*: the standard multiplication procedure, a multiplication assuming a server outside the chosen $2t+1$ servers that generate sub-shares has crashed or become maliciously corrupted, a multiplication assuming a server within the $2t+1$ set crashes, and a multiplication assuming a server within the set generates and sends incorrect sub-shares.

Figure 7.3 presents the latency results for four multiplication scenarios (default *Mult* protocol, *Mult* with an adversary outside the $2t+1$ sub-share generator server set, *Mult* with a crash, and *Mult* with invalid sub-share generation/distribution) assuming $n = 3t+1$ for thresholds for $t = 1$ (blue), 2 (orange), 3 (green). We note that latency introduced by timeouts during delayed or crashed server management is excluded from the analysis.

Results indicate that there is no observable difference between the standard multiplication and the scenario where a server outside the $2t+1$ set crashes or becomes corrupted, which is expected

as the corrupted server is not part of the selected $2t + 1$ responsible for generating sub-shares and thus not impedes the remaining $2t + 1$ servers from generating sub-shares and computing multiplication output shares.

Shifting to the third scenario where a server within the $2t + 1$ set crashes, the graph reports a total time of 169 ms for $t = 1$ which, given the average time to compute a multiplication under this setting is 70 ms, one can deduce the latency by breaking down the procedure in (a) an initial attempt at the multiplication which takes 70 ms (in reality is slightly shorter than 70 ms since the last step of generating shares is not completed) (b) an added 29 ms for detecting the crashed server and rolling back state (c) and a second attempt to complete the multiplication with the correct and updated set of servers (70 ms). This breakdown shows that the performance for $Mult$ assuming a server crash involves only a slight additional overhead, corresponding to an approximate 18% latency increase (assuming first and second multiplication attempts) for addressing the corrupted server. Similarly, for $t = 2$ and $t = 3$, the additional latencies observed are 53 ms (21% increase) and 87 ms (23% increase), respectively, showing a consistent and modest increase in overhead as the system scales.

In the last scenario where a byzantine server sends incorrect sub-shares, we have the total latency achieve $186, 316$ and 550 ms for $t = 1, 2, 3$, respectively. Assuming $t = 1$, the additional 46 ms observed can be attributed to the extra communication round needed to handle the accusation and the verification of the accused proposal sub-shares, which take 32% of the recovered multiplication total latency and take 34% and 37% for the remaining $t = 2$ and $t = 3$ configurations.

Chapter 8

Conclusion & Future Work

8.1 Conclusion

The interest in studying multi-party computation lies in its potential as an asset for secure data management across multiple entities, which has become predominant in today's cloud-dependent digital landscape. A key driving motivation of this work comes from recognizing that current frameworks often face a tradeoff between network setting, participant flexibility, and guarantees on computational output - challenges that must be addressed to advance the adoption of MPC in real-world industrial settings.

To tackle the problem of true practical MPC, this thesis introduces MPCServe, a novel dynamic MPC protocol designed to bring scalable and efficient multi-party computations to cloud-based environments. MPCServe employs a hybrid client-server model where the traditional MPC participants are replaced with clients that request computations to a set of flexible work servers. To enable guaranteed output under a partially-synchronous environment, we adapt an optimized version of the BGW MPC protocol [46] to fit the underlying confidential BFT-SMR layer while providing additional mechanisms to detect and handle delays, crashes, and Byzantine behavior, allowing servers to evaluate client-requested circuits without aborting. We also improve current state-of-art by defining mechanisms to distribute workload across a circuit's lifetime evaluation (Fairness) and provide sub-share forwarding to address servers that become unavailable to deliver sub-shares. We further employ a verifiable secret sharing (VSS) scheme based on Feldman's commitments together with non-interactive zero-knowledge proofs to ensure servers cannot distort the correctness of distributed sub-shares during multiplications. Performance analysis of MPCServe demonstrates promising efficiency, resolving multiplications at an average rate of $70, 116, 185$ milliseconds and large-scale circuits consisting of 10000 multiplications and 9999 additions at a reasonable processing speed of $12, 20, 39$ minutes, for realistic system configurations of $n = 3t+1$ and $1 \leq t \leq 3$, respectively.

8.2 Future Work

Given the complexity surrounding MPCServe, some aspects have been identified as future work and possible enhancements.

A promising concept is the integration of a generic circuit compiler to facilitate efficient function-to-circuit conversion, enabling any functionality to be fed and processed by the MPC system.

Another intriguing line of work involves improving the efficiency of the newly adopted mechanisms by exploring alternative approaches, namely replacing non-interactive zero-knowledge (NIZK) proofs with potentially faster zk-SNARKs [15], utilizing constant commitments [54] instead of linear Feldman commitments, and implementing methods to refresh shares within stored server proposals.

Additionally, while MPCServe includes a conceptual recovery mechanism for servers to obtain state information upon joining computations and talks about how shares during computations must be refreshed through DPSS to maintain computational correctness, the actual implementation of such procedures remains a core priority for future development.

Bibliography

[1] A. Aly, M. Keller, D. Rotaru, P. Scholl, N. Smart, and T.. Wood. *SCALE–MAMBA*, 2021.

[2] Frederik Armknecht, Colin Boyd, Kristian Gjøsteen, Angela Jäschke, Christian Reuter, and Martin Strand. A guide to fully homomorphic encryption. *Cryptology ePrint Archive*, 2015.

[3] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. An end-to-end system for large scale p2p mpc-as-a-service and low-bandwidth mpc for weak participants. *CCS '18: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 695–712, 2018.

[4] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. An end-to-end system for large scale p2p mpc-as-a-service and low-bandwidth mpc for weak participants. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, page 695–712, 2018.

[5] Soumya Basu, Alin Tomescu, Ittai Abraham, Dahlia Malkhi, Michael K Reiter, and Emin Gün Sirer. Efficient verifiable secret sharing with share recovery in bft protocols. *CCS '19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2387–2402, 2019.

[6] Donald Beaver. Efficient multiparty protocols using circuit randomization. *Advances in Cryptology—CRYPTO'91: Proceedings 11*, pages 420–432, 1991.

[7] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, pages 503–513, 1990.

[8] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[9] Assaf Ben-David, Noam Nisan, and Benny Pinkas. Fairplaymp: a system for secure multiparty computation. *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266, 2008.

[10] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. *STOC '93: Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, page 52–61, 1993.

[11] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). *Conference: Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 1–10, 1988.

[12] Daniel Bernhard, Orlando Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. *Advances in Cryptology – ASIACRYPT 2012. Lecture Notes in Computer Science, vol 7658*, pages 401–420, 2012.

[13] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the fiat-shamir heuristic and applications to helios. *Advances in Cryptology – ASIACRYPT 2012*, pages 626–643, 2012.

[14] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. State machine replication for the masses with bft-smart. *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

[15] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*, pages 326–349, 2012.

[16] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. *Proceedings of the twentieth annual ACM symposium on Theory of computing (STOC 1988)*, pages 103–112, 1988.

[17] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security, Volume 11, Issue 6*, page 403–418, November 2012 2012.

[18] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[19] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. Motion - a framework for mixed-protocol multi-party computation. *ACM Transactions on Privacy and Security*, Vol. 25(2):1–35, 2022.

[20] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. *Advances in Cryptology — CRYPTO 2001*, pages 524–541, 2001.

[21] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of CRYPTOLOGY*, 13:143–202, 2000.

[22] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, 1988.

[23] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. *26th Annual Symposium on Foundations of Computer Science (SFCS)*, pages 383–395, 1985.

[24] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. Fluid mpc: secure multiparty computation with dynamic participants. *Advances in Cryptology – CRYPTO 2021: 41st Annual International Cryptology Conference*, pages 94–123, 2021.

[25] Sebastian Coretti, Juan Garay, Martin Hirt, and Vassilis Zikas. Constant-round asynchronous multi-party computation based on one-way functions. *Advances in Cryptology – ASIACRYPT 2016*, pages 998–1021, 2016.

[26] Ronald Cramer, Ivan Damgård, and Ueli Maurer. General secure multi-party computation from any linear secret-sharing scheme. *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 316–334, 2000.

[27] Ronald Cramer and Ivan Damgård. Zero-knowledge proofs for finite field arithmetic, or: Can zero-knowledge be for free? *LNCS*, 1462, 07 1998.

[28] Nielsen JB Cramer R, Damgard IB. Secure multiparty computation and secret sharing. *Cambridge University Press*, 2015.

[29] Joan Daemen and Vincent Rijmen. Specification for the advanced encryption standard (aes). *FIPS PUB 197*, 2001.

[30] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. *CRYPTO 2012. Lecture Notes in Computer Science*, Vol. 7417:643–662, 2012.

[31] Ivan Damgård, Martin Geisler, Morten Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. *Public Key Cryptography – PKC 2009. Lecture Notes in Computer Science*, 5443:160–179, 2009.

[32] Giovanni Deligios, Aarushi Goel, and Chen-Da Liu-Zhang. Maximally-fluid MPC with guaranteed output delivery. *Cryptology ePrint Archive, Paper 2023/415*, 2023.

[33] Yvo Desmedt and Yair Frankel. Shared generation of authenticators and signatures. *CRYPTO '91. CRYPTO 1991. Lecture Notes in Computer Science*, Vol. 576:457–469, 1991.

[34] Yvo G Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, pages 449–458, 1994.

[35] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[36] Wenliang Du and Mikhail J. Atallah. Privacy-preserving cooperative statistical analysis. *Proceedings of the 17th Annual Computer Security Applications Conference*, 2001.

[37] Wenliang Du and Mikhail J. Atallah. Privacy-preserving statistical analysis. *Proceedings of the 17th Annual Computer Security Applications Conference*, December 2001.

[38] Sisi Duan, Xin Wang, and Zhang Haibin. Fin: Practical signature-free asynchronous common subset in constant time. *Cryptology ePrint Archive, Paper 2023/154*, 2023.

[39] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[40] Karim Eldefrawy, Tancrède Lepoint, and Antonin Leroux. Communication-efficient proactive mpc for dynamic groups with dishonest majorities. *International Conference on Applied Cryptography and Network Security*, pages 565–584, 2022.

[41] Karim Eldefrawy, Rafail Ostrovsky, Sunoo Park, and Moti Yung. Proactive secure multiparty computation with a dishonest majority. *Security and Cryptography for Networks: 11th International Conference*, pages 200–215, 2018.

[42] Karim Eldefrawy and Vitor Pereira. A high-assurance evaluator for machine-checked secure multiparty computation. *CCS '19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, page 851–868, 2019.

[43] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A pragmatic introduction to secure multi-party computation. *Foundations and Trends in Privacy and Security*, Vol. 2(2-3):70–246, 2018.

[44] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive, Paper 2012/144*, 2012.

[45] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. *28th Annual Symposium on Foundations of Computer Science (SFCS 1987)*, pages 427–438, 1987.

[46] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified vss and fast-track multiparty computations with applications to threshold cryptography. *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, page 101–111, 1998.

[47] Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. Yoso: You only speak once: Secure mpc with stateless ephemeral roles. *CRYPTO 2021: Advances in Cryptology*, pages 64–93, 2021.

[48] Oded Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, 78(110):1–108, 1998.

[49] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. *STOC '85: Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, page 291–304, 1985.

[50] Zhitao Guan, Xiao Zhou, Peng Liu, Longfei Wu, and Wenti Yang. A blockchain-based dual-side privacy-preserving multiparty computation scheme for edge-enabled smart grid. *IEEE Internet of Things Journal*, 9(16), 2022.

[51] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. *Advances in Cryptology - CRYPT0' 95*, 1995.

[52] Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Cryptographic asynchronous multi-party computation with optimal resilience. *Advances in Cryptology – EUROCRYPT 2005*, pages 322–340, 2005.

[53] Renuga Kanagavelu, Zengxiang Li, Juniarto Samsudin, Yechao Yang, Feng Yang, Rick Siow Mong Goh, Mervyn Cheah, Praewpiraya Wiwatphonthana, Khajonpong Akkarajitsakul, and Shangguang Wang. Two-phase multi-party computation enabled privacy-preserving federated learning. *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 410–419, 2020.

[54] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. *Advances in Cryptology - ASIACRYPT 2010*, pages 177–194, 12 2010.

[55] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. *CCS '20: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1575–1590, 2020.

[56] Marcel Keller, Peter Scholl, and Nigel P Smart. An architecture for practical actively secure mpc with dishonest majority. *CCS '13: Proceedings of the 2013 ACM SIGSAC conference on Computer and communications security*, pages 549–560, 2013.

[57] Brian Knott, Shobha Venkataraman, Awni Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. Crypten: Secure multi-party computation meets machine learning. *NIPS'21: Proceedings of the 35th International Conference on Neural Information Processing*, abs/2109.00984:4961–4973, 2021.

[58] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, JULY 1982.

[59] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. *Advances in Cryptology — CRYPTO 2000. CRYPTO 2000. Lecture Notes in Computer Science, vol 1880.*, pages 36–54, 2000.

[60] J. Liu, M. Juuti, Y. Lu, and N. Asokan. Oblivious neural network predictions via minionn transformations. *ACM CCS 17: 24th Conference on Computer and Communications Security*, pages 619–631, 2017.

[61] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. *STOC '12: Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234, 2012.

[62] N. Macon and A. Spitzbart. Inverses of vandermonde matrices. *The American Mathematical Monthly*, 65(2):95–100, February 1958.

[63] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. Churp: Dynamic-committee proactive secret sharing. *CCS '19: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[64] Bartosz Przydatek Martin Hirt, Ueli Maurer. Efficient secure multi-party computation. *Advances in Cryptology — ASIACRYPT 2000. Lecture Notes in Computer Science, vol 1976*, pages 143–161, December 2000.

[65] Silvio Micali, Oded Goldreich, and Avi Wigderson. How to play any mental game. *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229, 1987.

[66] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key fhe, advances in cryptology – eurocrypt 2016. *Lecture Notes in Computer Science, vol 9666*, pages 735–763, 2016.

[67] National Institute of Standards and Technology (NIST). Secure hash standard (shs). *FIPS PUB 180-4*, 2012.

[68] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks extended abstract. *PODC '91: Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, page 51–59, 1991.

[69] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. *Advances in Cryptology — CRYPTO '91*, pages 129–140, 1992.

[70] Michael O. Rabin. How to exchange secrets with oblivious transfer. *Aiken Computation Laboratory, Harvard University*, 1981.

[71] Michael O. Rabin. Randomized byzantine generals. *24th Annual Symposium on Foundations of Computer Science (SFCS)*, pages 403–409, 1983.

[72] Rahul Rachuri and Peter Scholl. Le mans: Dynamic and fluid mpc for dishonest majority. *CRYPTO 2022: Advances in Cryptology*, pages 719–749, 2022.

[73] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys.*, page 299–319, December 1990.

[74] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic. *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, page 148–164, 1999.

[75] David Schultz, Barbara Liskov, and Moses Liskov. Mpss: mobile proactive secret sharing. *ACM Transactions on Information and System Security (TISSEC)*, Vol. 13(4):1–32, 2010.

[76] Adi Shamir. How to share a secret. *Communications of the ACM*, Vol. 22:612–613, 1979.

[77] Bhavani Shankar, Kannan Srinathan, and C Pandu Rangan. Alternative protocols for generalized oblivious transfer. *Distributed Computing and Networking: 9th International Conference*, pages 304–309, 2008.

[78] Markus Stadler. Publicly verifiable secret sharing. *Advances in Cryptology - EUROCRYPT '96*, pages 190–199, 1996.

[79] Robin Vassantlal, Eduardo Alchieri, Bernardo Ferreira, and Alysson Bessani. Cobra: Dynamic proactive secret sharing for confidential bft services. *2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA*, pages 1335–1353, 2022.

[80] Edward Waring. Problems concerning interpolations. *Philosophical Transactions of the Royal Society*, 69:59–67, 1779.

[81] Yulin Wu, Xuan Wang, Willy Susilo, Guomin Yang, Zoe L Jiang, Siu-Ming Yiu, and Hao Wang. Generic server-aided secure multi-party computation in cloud computing. *Computer Standards and Interfaces*, 79:103552, 2022.

[82] Andrew C Yao. Protocols for secure computations. *23rd annual symposium on foundations of computer science (SFCS)*, pages 160–164, 1982.

[83] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. *SIGOPS Oper. Syst. Rev.*, page 253–267, October 2003.

[84] Wenxuan Yu, Minghui Xu, Bing Wu, Sisi Duan, and Xiuzhen Cheng. Admpc: Fully asynchronous dynamic mpc with guaranteed output delivery, 2024.