

2023-09-22

# Cryptalphabet Soup: DPFs meet MPC and ZKPs

Storrier, Kyle

---

Storrier, K. (2023). Cryptalphabet soup: DPFs meet MPC and ZKPs (Master's thesis, University of Calgary, Calgary, Canada). Retrieved from <https://prism.ucalgary.ca>.

<https://hdl.handle.net/1880/117318>

*Downloaded from PRISM Repository, University of Calgary*

UNIVERSITY OF CALGARY

Cryptalphabet Soup: DPFs meet MPC and ZKPs

by

Kyle Storrier

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES  
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE  
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

SEPTEMBER, 2023

© Kyle Storrier 2023

# Abstract

Secure multiparty computation (MPC) protocols enable multiple *parties* to collaborate on a computation using private inputs possessed by the different parties in the computation. At the same time, MPC protocols ensure that no participating party learns anything about the other parties' private inputs beyond what they can infer from the computation's output and their own inputs. MPC has wide ranging applications for privacy protecting systems. However, these systems have been plagued by limited performance, lack of scalability, and poor accuracy.

In this thesis, we demonstrate several novel techniques for using distributed point functions (DPFs) in combination with MPC to obtain significant performance improvements in several different applications. Namely, using novel observations about the structure of the most efficient available DPF construction in the literature, we show that DPF keys from untrusted sources can be checked for correctness using an MPC protocol between the two key holders, with direct applications in sender-anonymous messaging. We expand these observations to produce the most efficient available method to evaluate piecewise-polynomial functions, also known as splines. The scalability and efficiency of this method allows for splines to be used for extremely high accuracy approximation of non-linear functions in MPC. Furthermore, the protocols proposed in this thesis far outperform prior solutions both in large-scale asymptotic measurements and in concrete benchmarks using high-performance software implementations at both small- and large-scale.

# Preface

The results presented in this thesis include works from some published and unpublished work I have completed in collaboration with my supervisor, Dr. Ryan Henry, and several other collaborators. In particular, the material presented in [Chapter 4](#) and [Chapter 5](#) is drawn from the paper “GROTTO: Screaming fast  $(2 + 1)$ -PC for  $\mathbb{Z}_2^n$  via  $(2, 2)$ -DPFs”, a currently unpublished work which I collaborated on with Dr. Adithya Vadapalli, Allan Lyons, and Dr. Ryan Henry. The material about the Sabre sender-anonymous messaging system and DPF key verification is drawn from the paper “Sabre: Sender-Anonymous Messaging with Fast Audits” published at the 2022 IEEE Symposium on Security and Privacy which I collaborated with Dr. Adithya Vadapalli and Dr. Ryan Henry on. The information from [Chapter 6](#) and the sections on MPC generation of DPF keys from [Chapter 7](#) are part of an ongoing research project which I am collaborating with Dr. Adithya Vadapalli, Allan Lyons, and Dr. Ryan Henry on.

# Acknowledgments

I would like to express my sincere gratitude to everyone who helped me throughout my graduate studies. First among these is my MSc supervisor, Dr. Ryan Henry, whose experience, patience, and flexibility has helped me immeasurably throughout this experience. Ryan's guidance has given me opportunities to grow and develop in my research and technical skills and also in my communication, presentation, writing, and teaching abilities as I have navigated the process of researching, writing, publishing, and presenting papers, developing this thesis, and working as a teaching assistant. I am thankful for each and every one of these opportunities, and for Ryan's dedication to helping me through each of these steps.

I am also grateful for the students and researchers I had the opportunity to meet through the Digital Liberties Lab at the University of Calgary. These connections have been invaluable to me both academically and personally. In particular I want to thank Adithya Vadapalli and Allan Lyons for collaborating with me on the much of the work which appears in this thesis. I also want to thank Chris Jiang for his work on `dpf++`.

Finally, I would like to thank my family. The support of my parents, Dave and Nadine Storrier, has been a blessing throughout my graduate studies. My siblings and our dog, Daisy, have also been a welcome source of company and support for which I am grateful.

To mom and dad.

# Table of Contents

<b>Abstract</b>	<b>ii</b>
<b>Preface</b>	<b>iii</b>
<b>Acknowledgments</b>	<b>iv</b>
<b>Dedication</b>	<b>v</b>
<b>Table of Contents</b>	<b>vi</b>
<b>List of Tables</b>	<b>ix</b>
<b>List of Figures</b>	<b>x</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background and motivation	1
1.2 Thesis statement	2
1.3 Research contributions	3
1.4 Prior works	5
1.4.1 GROTTO prior works	5
1.4.2 Sabre prior works	7
1.5 Thesis layout	8
<b>2 Cryptographic preliminaries</b>	<b>9</b>
2.1 Mathematical notation	9
2.2 Fixed-point arithmetic	10
2.2.1 ULP error	12
2.3 Secret sharing	13
2.3.1 $(2, 2)$ -Additive sharing	13
2.4 Zero-knowledge proofs	14
2.5 Secure Multiparty Computation (MPC)	15
2.5.1 Secret share-based MPC	18
2.5.2 Garbled circuit-based MPC	19
2.5.3 Beaver multiplication triples	21
2.5.4 The ABY framework	24

2.6	Selection vector . . . . .	25
2.7	Point functions . . . . .	25
2.7.1	Distributed point functions . . . . .	26
2.8	Private information retrieval (PIR) . . . . .	27
2.8.1	Private information retrieval (PIR) from selection vectors . . . . .	27
2.8.2	DPF-based PIR . . . . .	29
2.8.3	DPFs for private writing . . . . .	30
2.9	LowMC block cipher . . . . .	32
<b>3</b>	<b>Point function trees and the Boyle–Gilboa–Ishai DPF construction</b>	<b>33</b>
3.1	Point function trees . . . . .	33
3.1.1	0/1-leaves and 0/1-nodes . . . . .	34
3.2	Boyle–Gilboa–Ishai DPF construction . . . . .	35
3.2.1	Constructing 0-pairs . . . . .	37
3.2.2	Constructing 1-pairs . . . . .	38
3.2.3	Chaining 1-pairs . . . . .	39
<b>4</b>	<b>Parity-segment trees and DPFs</b>	<b>43</b>
4.1	Parity-segment trees . . . . .	43
4.1.1	Computing segment parities . . . . .	44
4.1.2	Analysis . . . . .	49
4.2	Selection vector rotation . . . . .	52
4.2.1	Scalar-to-selection vector share conversion . . . . .	53
4.3	DPFs as parity-segment trees . . . . .	54
<b>5</b>	<b>Grotto: MPC via (2, 2)-DPFs</b>	<b>57</b>
5.0.1	The shoulders GROTTO stands upon . . . . .	58
5.0.2	Threat model . . . . .	59
5.0.3	Roadmap . . . . .	60
5.1	Technical Overview . . . . .	60
5.2	Function evaluation via PIR . . . . .	61
5.2.1	Fractional-bit reduction for shared secrets . . . . .	64
5.3	The Grotto framework . . . . .	70
5.3.1	Sign-corrected polynomial evaluation . . . . .	71
5.4	Implementation & evaluation . . . . .	77



5.5	Conclusion . . . . .	82
<b>6</b>	<b>Beyond GROTTO: DPFs for bit decomposition in MPC . . . . .</b>	<b>83</b>
6.1	Function composition . . . . .	83
6.2	Bit extraction . . . . .	86
6.3	Bit decomposition . . . . .	87
6.4	Performance evaluation . . . . .	89
6.5	Conclusion . . . . .	91
<b>7</b>	<b>MPC for DPF verification and generation . . . . .</b>	<b>92</b>
7.1	Sender-anonymous messaging preliminaries . . . . .	92
7.1.1	PIR-writing for SAM protocols . . . . .	93
7.1.2	DPF key correctness verification . . . . .	93
7.1.3	Sender-anonymous bulletin boards . . . . .	94
7.1.4	Sender-anonymous mailboxes . . . . .	95
7.1.5	Relationship with onion routing and mix networks . . . . .	95
7.2	MPC DPF key verification . . . . .	96
7.2.1	Secure (2+1)-party auditing . . . . .	96
7.3	MPC-in-the-head ZKPoK verification . . . . .	101
7.4	DPF key generation in MPC . . . . .	105
7.5	Sabre sender-anonymous messaging . . . . .	107
7.5.1	System Design . . . . .	108
7.5.2	Verifying Mailbox Addresses . . . . .	111
7.5.3	Security Guarantees . . . . .	112
7.6	Implementation and Evaluation . . . . .	112
7.6.1	Communication Cost . . . . .	113
7.6.2	Auditing . . . . .	113
7.6.3	Resistance to denial-of-service attack . . . . .	115
7.6.4	Head-to-head with Riposte and Express . . . . .	116
7.7	Conclusion . . . . .	120
<b>8</b>	<b>Conclusion . . . . .</b>	<b>121</b>
8.1	GROTTO related work . . . . .	121
8.2	Sabre related work . . . . .	122
8.3	Future Work . . . . .	123

8.4 Conclusion . . . . .	125
<b>Bibliography . . . . .</b>	<b>126</b>
<b>A Prefix-parity amortization via memoization . . . . .</b>	<b>134</b>
<b>B Formulae for (2+1)-PC sign-corrected polynomial evaluation . . . . .</b>	<b>136</b>
B.1 One-round ABY2.0-like evaluation . . . . .	137
B.1.1 For constant polynomials . . . . .	137
B.1.2 For linear polynomials . . . . .	137
B.1.3 For quadratic polynomials . . . . .	138
B.1.4 For cubic polynomials . . . . .	140
B.2 Two-round ABY2.0-like evaluation . . . . .	142
B.2.1 For constant polynomials . . . . .	142
B.2.2 For linear polynomials . . . . .	142
B.2.3 For quadratic polynomials . . . . .	142
B.2.4 For cubic polynomials . . . . .	143
B.3 Horner’s Method evaluation . . . . .	145
B.3.1 For linear polynomials . . . . .	145
B.3.2 For quadratic polynomials . . . . .	145
B.3.3 For cubic polynomials . . . . .	146
<b>C Proof of Theorem 2 . . . . .</b>	<b>147</b>
<b>D Proof of Theorem 3 . . . . .</b>	<b>148</b>
<b>E Stepping stones to Sabre-M . . . . .</b>	<b>150</b>
<b>F Sabre Security Analysis . . . . .</b>	<b>154</b>
<b>G LowMC parameter selection . . . . .</b>	<b>160</b>
<b>H Sabre Communication Cost . . . . .</b>	<b>162</b>

# List of Tables

1.1	Common abbreviations used throughout this work . . . . .	5
4.1	Prefix-parity computation sequence . . . . .	48
5.1	Summary of GROTTO gadgets . . . . .	78
5.2	GROTTO versus LLAMA cost comparison . . . . .	80
6.1	Secret Share Conversion Performance Comparison . . . . .	90
7.1	Sabre client communication cost . . . . .	105
7.2	Sabre server communication cost . . . . .	105
B.1	MPC polynomial evaluation cost comparison . . . . .	136

# List of Figures

2.1	Two server PIR using selection vectors . . . . .	28
3.1	Binary-tree representation for the 45th point function . . . . .	35
3.2	BGI tree shares for the DPF encoding the 27th point function . . . . .	36
4.1	Example parity-segment tree for a 64-bit string . . . . .	44
4.2	Prefix-parity algorithm pseudocode . . . . .	47
4.3	Selection vector rotation protocol . . . . .	53
4.4	Equivalence between rotating selection vectors versus databases . . . . .	54
6.1	Sequential versus parallel composition . . . . .	85
6.2	Bucket layout for bit extraction in GROTTTO . . . . .	86
7.1	Conditional swap functionality . . . . .	97
7.2	Merkle-tree commitment for 2-verifier SNIPs . . . . .	104
7.3	Binary-tree representation of a generalized point function . . . . .	108
7.4	Time to audit a batch of 128 requests in Sabre . . . . .	114
7.5	Sender-anonymous messaging auditing comparison . . . . .	115
7.6	Sabre 2-verifier SNIP auditing versus malformed request sampling . . . . .	117
7.7	Sabre variants, Riposte, and Express comparison . . . . .	118
7.8	Malformed queries performance comparison . . . . .	119
A.1	Prefix-parity empirical amortized costs . . . . .	135
E.1	Throughput of Sabre-M variants . . . . .	153
G.1	LowMC parameter evaluation . . . . .	161

# Chapter 1

## Introduction

### 1.1 Background and motivation

The ever increasing use of the internet and computers has driven wave upon wave of technological progress in a vast array of fields including research, communication, collaboration, entertainment, and education. However, this advancement has come at a cost. Privacy has been trampled by the rapid growth of theft and extortion by cybercriminals, invasive behavioural advertising by corporations, and surveillance by oppressive governments. Technological safeguards have struggled to keep up with the incessant growth in the depth and breadth of data being collected about internet users and the equally ceaseless development of the machine learning and data analytics tools required to process and exploit that data.

The response to these pervasive invasions of privacy comes in the form of *privacy-enhancing technologies (PETs)*. PETs are a diverse assortment of techniques, systems, and mechanisms designed to give users control over how their data is collected, shared, and used, through the use and application of advanced cryptographic primitives. This enables the development of *privacy-friendly* alternatives to existing, *privacy-agnostic*, technologies.

Secure multiparty computation (MPC) is one such component within PETs. Intuitively, MPC allows multiple parties to work together to perform a computation over private inputs held by various participating parties. This computation is done with the added privacy requirement that an honest party in the protocol will not

leak any information about their input to other parties, beyond the information those parties could already compute from their own private inputs and the protocol's output. While existing MPC frameworks are capable of performing any computation, most applications are unwilling to accept the significant increase in time required for MPC protocols, which is due, in large part, to the relatively slow movement of data between remote systems over a network. This limitation has led to extensive research being performed on improving the performance of MPC systems through new methodologies, hybrid frameworks which link together multiple existing systems, and other innovations.

The distributed point function (DPF) is a recently developed cryptographic primitive that has seen significant use in the PETs literature. DPFs were originally developed and formalized in the literature on privately reading and writing to databases, referred to as private information retrieval (PIR) and PIR-writing respectively [29]. Later works generalized on the DPF to create the broader concept of function secret sharing (FSS). The applications for DPFs have also been extended for use in MPC. Some of these extensions used DPFs directly [60, 62], but others use different FSS primitives, primarily the distributed comparison function (DCF) [11, 12, 14, 35].

## 1.2 Thesis statement

This thesis considers the intersection of DPF and MPC-based PETs with the aim of reducing the overall communication and computation cost for PETs based upon these techniques. Both DPFs and MPC are extremely popular in the PETs literature, with MPC having a long history in PETs and DPFs rapidly growing in popularity since their formalization in 2014. However, even after considerable development and research, the limiting factor on PETs employing these techniques is still efficiency. In the following chapters, this thesis introduces multiple protocols that combine MPC with DPFs to significantly improve the performance of PETs based upon DPFs and PETs based upon MPC, both asymptotically and concretely.

The primary technical contributions of this thesis fall into two categories, techniques for *fast MPC through leveraging DPFs* and techniques for *efficiently processing DPF keys in MPC*. In the case where DPFs are used to improve MPC protocols, the primary focus is on how DPFs can be precomputed and then used in additive secret

share-based MPC to allow complicated non-linear functions to be computed easily. These techniques primarily exploit the traditional DPF application of private information retrieval along with the unique structure of a DPF key to enable these optimizations. At the same time, the protocols for processing DPF keys in MPC are primarily focused on how a new pair of DPF keys can be generated and how to prove the correctness of DPF keys received from an untrusted source. In both cases, the structure of the DPF key and the DPF key generation algorithm forms a clear basis to enabling these techniques. These techniques can be applied in a large number of contexts, including for sender-anonymous messaging and as a key component in privacy preserving machine learning. In each of these cases, the techniques introduced here improves on the state-of-the-art solutions enabling faster and more scalable PETs both in theoretical analysis and in experimental studies.

### 1.3 Research contributions

This section summarizes the primary research contributions presented in the following chapters.

1. **Parity-segment trees and the prefix parity algorithm.** [Chapter 4](#) introduces the concept of a parity-segment tree, a variant of the existing segment tree data structure specifically designed for parity queries over a bitstring. This data structure is closely related to the prefix-parity algorithm introduced in [Section 4.1.1](#). The prefix-parity algorithm provides an efficient method to determine the parity of all intervals specified by an arbitrary partition of a bitstring, represented as a prefix-segment tree.
2. **Formal characterization of Boyle–Gilboa–Ishai DPFs.** [Chapter 3](#) introduces the DPF construction of Boyle, Gilboa, and Ishai [14], and provides a novel characterization of this DPF construction to clarify the properties of these DPFs. In particular, by classifying DPF nodes into 0-nodes and 1-nodes, it becomes clear that DPF keys can act as secret shared parity-segment trees, as is shown in [Section 4.3](#). It is also shown in [Section 4.2](#) that, by viewing DPFs as compressed forms of selection vectors, it is possible to efficiently “rotate”

precomputed DPFs to have a new, secret shared, distinguished point chosen at runtime.

3. **Improved protocols for fixed-point arithmetic.** [Section 5.3.1](#) introduces a novel protocol for performing MPC polynomial evaluation using one round of communication. It also introduces new *probabilistic* and *exact* fractional precision–reduction procedures for signed fixed-point numbers
4. **Use Boyle–Gilboa–Ishai DPFs to create the GROTTO framework.** GROTTO evaluates non-linear functions by using spline functions to approximate the given function with sufficiently high accuracy that, in many cases, the “approximation” is as close to the exact answer as is possible for a fixed-point numbers being used. The spline evaluation step leverages both the parity-segment tree structure of Boyle–Gilboa–Ishai DPFs and the improved fixed-point polynomial evaluation techniques from [Section 5.3.1](#).
5. **Protocol for MPC verification of DPF key correctness.** [Section 7.2](#) describes a novel MPC protocol to verify that a Boyle–Gilboa–Ishai DPF is well-formed. This protocol traverses down from the root of the DPF tree to the leaf node corresponding to the distinguished point for that DPF. At each level of the tree, the parties can exploit the characteristics of Boyle–Gilboa–Ishai DPFs in order to check the correctness at that level. Having this property hold for every node on the path from the root of the tree to the distinguished point’s leaf is sufficient to prove that the DPF key pair is valid. We use this protocol to create the Sabre system
6. **Protocol for MPC generation of DPF keys.** [Section 7.2](#) observes that the MPC protocol for verifying DPF keys shares a common structure with the algorithm for generating DPF key pairs. The clear application of this idea is to obliviously generate DPFs with secret shared distinguished points. This is especially important for adapting MPC systems that use DPFs, such as GROTTO, to work in two-party settings where there is no third party to provide precomputed DPF keys.



Abbreviation	Definition
MPC	Secure Multiparty
ZKP	Zero-Knowledge Proof
ZKPoK	Zero-Knowledge Proof of Knowledge
PIR	Private Information Retrieval
DPF	Distributed Point Function
DCF	Distributed Comparison Function
FSS	Function Secret Sharing
LUT	Lookup Table
SAM	Sender-Anonymous Messaging
PET	Privacy-Enhancing Technology
DoS	Denial-of-Service
OT	Oblivious Transfer
AHE	Arithmetic Homomorphic Encryption

Table 1.1: Common abbreviations used throughout this work

## 1.4 Prior works

Because GROTTO and Sabre explore the relationship between MPC and DPFs in two different settings, the prior works related to these systems differs substantially.

### 1.4.1 GROTTO prior works

In the related works that are most closely related to GROTTO, there are two primary directions that have been pursued.

**DPFs with full-domain evaluation.** The two examples of this approach are Pirsona [60] and Pika [62]. In these systems, the primary goal is to convert the computation into a PIR query. To do this, the parties need to first produce a DPF whose distinguished point is equal to the given input  $x$ . In order to generate this DPF while keeping  $x$  private, both of these systems employ the technique of shifting the outputs of the DPF to change the distinguished point, which we discuss in [Section 4.2](#). However, neither of these works employ the technique of rotating the database being queried backwards. In both of these existing protocols, the two parties each hold a copy of a precomputed database, which we refer to as a lookup table (LUT). Each party can use the

DPF key it possesses to perform a PIR query against the LUT. In Pika, index  $i$  in the LUT contains the outputs of the function being computed at the corresponding input value. Pirsona uses the same technique for computing integer comparisons, but for computing the inverse square root of fixed-point numbers Pirsona’s LUT contains the coefficients for a linear approximation of the function near the given input value. This allows a lower precision input value to be used for the PIR query, reducing the cost of the PIR query and the size of the DPF key and the LUT, while also allowing a higher precision version of the same input to be used when evaluating the linear approximation. In both of these systems, the primary performance bottleneck is due to the PIR query which requires an expensive full-domain evaluation of the DPF.

**DCF and other FSS primitives.** FSS was developed as an extension of DPFs with the goal of creating secret-shared forms of functions other than point functions [11, 12, 14]. The most relevant of these FSS primitives is the DCF, which is the secret shared form of the less-than comparison function. These, in turn, can be used to create FSS schemes for tasks such as bit-decomposition and interval containment, which can itself be used for spline evaluation [11]. Boyle et al. show that these FSS schemes can be used directly for MPC applications [11]. In contrast, the LLAMA system, by Gupta et al., uses DCFs to retrieve polynomial coefficients for a spline function approximating the non-linear function being evaluated, similar to what we describe in Section 5.2. They can then evaluate the retrieved polynomial. In both of these cases, the added complexity of the more complex FSS schemes reduces the system’s overall efficiency. In particular, the key sizes of these FSS schemes is much higher than for a DPF. At the same time, the computational cost, while better than many DPF-based techniques, is still quite high in LLAMA.

Clearly, existing MPC techniques based upon DPFs and more general FSS are extremely effective at evaluating non-linear functions, either exactly or approximately depending on the system. However, high computation costs for DPF-based systems and large keys in FSS-based systems limits the practicality of these systems. In Chapter 4 and Chapter 5, we show how GROTTO can combine DPFs with our novel

prefix-parity algorithm in order to achieve performance that is faster than any prior solution using only DPF keys.

#### 1.4.2 Sabre prior works

There are two primary prior works on sender-anonymous messaging (SAM) protocols. The first of these is Riposte [21], a system in the sender-anonymous bulletin board model of SAM protocols. The other is Express, which is in the sender-anonymous mailbox model [25]. The primary difference between these two models is that the bulletin board model allows for anonymous posting to a public location, like an anonymous version of Twitter, whereas the mailbox model allows messages to be sent anonymously to an individual user’s mailbox by any sender that knows that user’s secret mailbox address.

In both cases, the SAM protocol is based on using PIR-writing with DPFs to perform message writes. Because the writes are being performed by a potentially malicious client, the servers in these systems need a way to verify the correctness of the DPF keys they receive. Riposte can use a three-party MPC protocol or a zero-knowledge proof (ZKP) to verify the DPFs it uses. However, both methods require the use of a DPF with keys of size  $O(\sqrt{N})$  [21] rather than the Boyle–Gilboa–Ishai DPFs, which have keys of size  $O(\log N)$  [14]. Additionally, the ZKP-based solution, which can tolerate  $s - 1$  out of  $s$  malicious servers, requires the use of a computationally expensive *seed-homomorphic pseudorandom generator*. Express takes a different approach by using the logarithmic size Boyle–Gilboa–Ishai DPFs in combination with a two-server auditing protocol that requires  $O(1)$  rounds of communication and a full-domain evaluation of the DPF being audited, which has computational cost of  $O(N)$  [25]. This protocol works very well when the queries being received are primarily valid, as the full-domain evaluation is required for the PIR-writing step anyway. However, performing the full-domain evaluation as part of the audit makes Express vulnerable to a denial of service attack which overwhelms the server with invalid DPFs that it needs to audit.

Sabre addresses the shortcomings of both Riposte and Express. In particular, Sabre uses the logarithmic size Boyle–Gilboa–Ishai DPFs that are used in Express. In order to make the auditing step more resistant to denial of service attacks, Sabre em-

employs a  $(2 + 1)$ -party MPC auditing protocol, which does not require the full-domain evaluation of the DPF. This protocol has computation cost and round complexity  $O(\log N)$ . It can also be converted into a zero-knowledge proof of knowledge (ZKPoK) through the use of our novel variant of the MPC-in-the-head paradigm of Ishai, Kushilevitz, Ostrovsky, and Sahai [39], which we describe in [Section 7.3](#). The mailbox model version of Sabre, called Sabre-M, also uses a highly efficient mailbox address verification process, which we describe in [Section 7.5.2](#).

## 1.5 Thesis layout

The remainder of this thesis describes the way in which we address the problems and shortcomings in prior works. In [Chapter 2](#), we present the relevant cryptographic preliminary material. The Boyle–Gilboa–Ishai DPF construction is presented and characterized in detail in [Chapter 3](#), and in [Chapter 4](#) we detail how these DPFs can be used as a novel data structure, known as a parity-segment tree. [Chapter 5](#) shows how the structure of Boyle–Gilboa–Ishai DPFs can be used to create the GROTT system for fast non-linear MPC. This system is expanded to enable efficient bit decomposition in [Chapter 6](#). The Sabre system and its techniques for DPF verification are described in [Chapter 7](#), along with related techniques for DPF key generation in MPC. The conclusions and results are summarized in [Chapter 8](#).

## Chapter 2

# Cryptographic preliminaries

In this chapter we introduce the notation, concepts, and prior works which serve as the building blocks for the major contributions presented in the subsequent chapters.

### 2.1 Mathematical notation

We deal extensively with vectors over  $\mathbb{Z}_N$ . For the special case where  $N = 2$ , we equate such a vector with the corresponding bitstring (i.e., the bitstring composed of the same bits, in the same order). We use ' $\oplus$ ' to denote the bitwise exclusive-OR (XOR) operator and '+' to denote normal addition in a ring (or a module over a ring) of characteristic other than 2.

We write  $\gg$  and  $\ll$  respectively for the arithmetic (sign-extended) right shift and logical left shift applied to fixed-width bitstrings; likewise, we write  $\ggg$  and  $\lll$  for *cyclic rotation* to the right and left, whether of a bitstring or of a vector. When the cyclic rotation is by a negative distance  $i$ , we define the rotation to be a distance  $-i$  rotation in the opposite direction or, equivalently, a rotation in the original direction by the smallest positive integer congruent to  $i$  modulo the bitstring length,  $N$ .

Let  $s$  be a bitstring. We denote the *substring* of  $s$  starting at index  $a$  (inclusive) and ending at  $b$  (exclusive) by  $s[a..b)$ . We call a substring  $s[0..b)$  with starting index 0 a *prefix* of  $s$ .

Some of the primitives and constructions considered herein provide *computa-*

*tional* security guarantees. In such instances, a special value  $\lambda \in \mathbb{N}$  called a *security parameter* acts as a tuning knob, enabling defenders to select efficiency–security tradeoffs they deem palatable. Loosely speaking, the resources required of “honest” players are bounded by some (reasonably small) polynomial  $p(\lambda)$ , whereas an attacker whose resources are bounded by *any* polynomial in  $\lambda$ —even an astronomically large one—will provably succeed in compromising security with at most some “insignificant” probability.

More precisely, the success of the attacker in such cases is described by a so-called *negligible function* [31, 43].

**Definition 1.** A function  $\mathbb{N} \rightarrow \mathbb{R}^{\geq 0}$  is *negligible* if  $\varepsilon(\lambda) \in o(\lambda^{-a})$  for every  $a \in \mathbb{N}$ .

Intuitively, a function  $\varepsilon$  being negligible means that it vanishes asymptotically faster than any positive inverse polynomial; hence, by making the honest parties expend *slightly* more resources, we can force attackers to expend *super-polynomially* more resources in order to maintain the same successful attack probability. Often, we find ourselves concerned with the probability that some undesirable event *does not* happen; if  $\varepsilon(\text{security})$  is negligible, then its complement  $1 - \varepsilon(\lambda)$  is called *overwhelming*.

As is customary, we model both defenders and attackers alike as probabilistic polynomial-time (PPT) algorithms, and we pass the security parameter to these algorithms in unary  $1^\lambda$  for consistency with traditional input length–based characterizations of running times.

Given a finite set  $S$ , we use  $a \in_{\mathbb{R}} S$  to denote the uniform random selection of  $a$  from  $S$ .

A pseudorandom generator (PRG) is a cryptographic primitive which takes a short random “seed” and uses it to generate some number of pseudorandom bits, which are computationally indistinguishable from random bits.

## 2.2 Fixed-point arithmetic

Fixed-point representations encode (approximations to) real numbers using signed integers in two’s-complement format [65]. Specifically, the fixed-point approximation to  $x \in \mathbb{R}$  is  $\lfloor x \cdot 2^p \rfloor$ , where  $p \in \mathbb{N}$  is a *fractional precision* parameter indicating how

many bits to reserve for the fractional (non-integer) part of  $x$ . Assuming 64-bit representations, this leaves  $64 - p - 1$  bits for the integer part of  $x$  (plus one bit for the sign).

For example, the fixed-point approximation to  $\pi = 3.14159\dots$  using  $p = 16$  fractional bits is

$$\begin{aligned} \lfloor \pi \cdot 2^{16} \rfloor &= \lfloor 205\,887.41614566\dots \rfloor \\ &= 205\,887 \quad \text{fractional part} \\ &= 0x\underbrace{00000000000000}_{\text{sign bit + integer part}}\overbrace{3243f}^{\text{fractional part}}. \end{aligned}$$

Addition (or subtraction) of fixed-point numbers (assuming a common  $p$ ) is realized using addition (or subtraction) of the underlying integers. The resulting sum (or difference) is exact, provided no overflow occurs.

To multiply fixed-point numbers  $x_0$  and  $x_1$ , respectively having  $p_0$  and  $p_1$  fractional bits, it suffices to multiply the underlying integer representations. The resulting product has  $p = p_0 + p_1$  fractional bits and is likewise exact when no overflow occurs [65].

For example, we can compute the area of a circle with (unitless) radius  $r = 1.25$  by expressing  $r$  as a fixed-point number and computing

$$\begin{aligned} \lfloor \pi \cdot 2^{16} \rfloor \cdot \lfloor 1.25 \cdot 2^{16} \rfloor^2 &= 205\,887 \cdot 81\,920^2 \\ &= 1\,381\,684\,268\,236\,800 \\ &= 0x\underbrace{00004e8a27000000}_{\text{sign bit + integer part}}\overbrace{0000}^{\text{fractional part}}, \end{aligned}$$

a fixed-point number with  $p' = 16 + (16 + 16) = 48$  fractional bits and, consequently, just  $64 - 48 - 1 = 15$  bits remaining for the integer part.

To “reset” the number of fractional bits back to  $p = 16$ , it suffices to perform an arithmetic (sign-extending) right shift by  $p' - p = 32$  bits; that is,

$$\begin{aligned} (0x\underbrace{00004e8a27000000}_{\text{sign bit + integer part}} \gg 32) &= 0x\underbrace{000000000000}_{\text{sign bit + integer part}}\overbrace{0004e8a2}^{\text{fractional part}} \quad (2) \\ &= 321\,698 \\ &= \lfloor 4.908721923828125 \cdot 2^{16} \rfloor. \end{aligned}$$

Meanwhile,  $\pi \cdot 1.25^2 = 4.9087385 \dots$  so that

$$\pi \cdot r^2 - 4.908721923828125 \approx 0.0000165974059.$$

This technique demonstrates the more general phenomenon that right shifting a fixed-point number reduces that number's precision. Similarly, left shifting increases precision without increasing the accuracy of the resulting approximation.

### 2.2.1 ULP error

It is impossible to perfectly represent all real numbers using a finite number of bits. As a result, floating-point and fixed-point computations often deviate from the exact real-number result for that computation. A common metric used to evaluate the accuracy of a mathematical computation being performed by a computer is *units in the last place (ULP) error* [30].

The idea behind ULP error, in the case of fixed-point numbers, is to suppose that  $r_1 \in \mathbb{R}$  is the exact result of a given computation, and that  $r_2$  is the approximate fixed-point result produced by the actual computation. The ULP error is then defined as being the number of representable fixed-point values between  $r_1$  and  $r_2$ . The same idea is generally applied to floating-point numbers, but we focus specifically on the case of fixed-point numbers.

For example, consider the fixed-point computation for the area of a circle shown previously in Equation (2). Here, the difference between the real solution and its fixed-point approximation is approximately 0.0000165974059 which is greater than  $2^{-16}$  and less than  $2 \cdot 2^{-16}$ . Since  $2^{-16}$  is the smallest difference between two fixed-point numbers with 16 bits of fractional precision, there is one representable fixed-point values between the true solution and the fixed-point approximation, so this computation has 1 ULP of error. We can also see this by looking at the integer value used to represent the fixed-point numbers. The approximated fixed-point calculation produces an underlying integer 321698, whereas the fixed-point approximation of the exact solution is  $\lfloor 4.9087385 \dots \cdot 2^{16} \rfloor = 321699$ . Similarly, if the fixed-point precision is reduced by performing an arithmetic right shift by one bit, then the fixed-point calculation produces a fixed-point number with an underlying integer  $\lfloor 4.908721923828125 \cdot 2^{15} \rfloor = 160849$  and the exact solution become



$\lfloor 4.9087385 \dots \cdot 2^{15} \rfloor = 160\,849$ . Since these two results are the same, this result, with 15 bits of precision in the answers, has 0 ULPs of error. Conversely, we can increase the precision of the fixed point number from 16 fractional bits to 17 fractional bits by shifting the bits to the left by four places, this does not change the accuracy of the fixed-point number but it does increase the precision. The resulting fixed-point number, computed at 16 bits of precision but increased to 17 bits of precision afterward, has an underlying integer value 643 396 to represent 4.908721923828125. The exact solution represented as a fixed-point number with 17 bits of fractional precision has an underlying integer value  $\lfloor 4.9087385 \dots \cdot 2^{17} \rfloor = 643\,398$ . As a result, this approximation, using 17 bits of precision, has 2 ULPs of error.

## 2.3 Secret sharing

Secret sharing allows a *dealer* to distribute a secret among two or more *shareholders* in such a way that individual shareholders learn nothing while “authorized subsets” of shareholders easily learn the whole secret [57]. A common form of secret sharing is  $(t, n)$ -threshold secret sharing. In this scenario, which was independently formalized by Shamir [57] and Blakley [9], there are  $n$  shareholders and any subset of  $t$  or more among them is authorized to learn the secret. Various threshold secret-sharing techniques have been developed, but this work focuses on simple “additive” schemes wherein secret reconstruction is accomplished via addition (or XOR) [40]. These schemes, described in the following sections, are especially popular in MPC applications. Unless otherwise stated, all shares in this thesis are assumed to be 64-bit  $(2, 2)$ -threshold shares. This means that the dealer splits the secret into  $n = 2$  shares which it then sends to the  $n = 2$  shareholders, and both ( $k = 2$ ) shareholders need to work together in order to reconstruct the secret value.

### 2.3.1 $(2, 2)$ -Additive sharing

A  $(2, 2)$ -additive sharing of a number  $a \in \mathbb{Z}[n]$  is denoted  $[a] = ([a]_0, [a]_1) \in \mathbb{Z}[n] \times \mathbb{Z}[n]$ , such that  $[a]_0 + [a]_1 = a$ .

To share a 64-bit integer  $S$ , the dealer samples  $[S]_0$  uniformly at random, sets  $[S]_1 \leftarrow S - [S]_0 \bmod 2^{64}$ , and then sends  $[S]_b$  to shareholder  $b$  for  $b = 0, 1$ . To recover

$S$ , the shareholders pool their shares and compute

$$\begin{aligned} [S]_0 + [S]_1 &\equiv [S]_0 + (S - [S]_0) \\ &\equiv S \pmod{2^{64}}. \end{aligned}$$

Additive secret sharing is linearly homomorphic: Given additive sharings  $[S] := ([S]_0, [S]_1)$  and  $[T] := ([T]_0, [T]_1)$  alongside non-secret scalars  $c$  and  $d$ ,

$$\begin{aligned} (c \cdot [S]_0 + d \cdot [T]_0) + (c \cdot [S]_1 + d \cdot [T]_1) &= c \cdot ([S]_0 + [S]_1) \\ &\quad + d \cdot ([T]_0 + [T]_1) \\ &\equiv c \cdot S + d \cdot T \pmod{2^{64}}, \end{aligned}$$

so that  $(c \cdot [S]_0 + d \cdot [T]_0, c \cdot [S]_1 + d \cdot [T]_1)$  is a  $(2, 2)$ -additive sharing of the linear combination  $c \cdot S + d \cdot T$ . Notice that shareholder  $b$  can compute  $[c \cdot S + d \cdot T]_b := c \cdot [S]_b + d \cdot [T]_b$  locally—i.e., without interacting with its peer. Consequently, they can collaborate to learn the linear combination  $c \cdot S + d \cdot T$  without learning anything extra about  $S$  or  $T$ .

### **(2, 2)-XOR sharing**

XOR-shares are just  $n$ -tuples of additive shares over  $\mathbb{Z}_2$ . In this setting, “addition” is just the bitwise exclusive-OR operator and, by convention, “multiplication” is the bitwise logical-AND operator. We write  $\langle x \rangle = (\langle x \rangle_0, \langle x \rangle_1)$  to denote an XOR-sharing of  $x$  (in contrast with writing  $[x] = ([x]_0, [x]_1)$  for an additive sharing of the same).

## **2.4 Zero-knowledge proofs**

A zero-knowledge proof of knowledge (ZKPoK) is a powerful cryptographic primitive with which one party, known as a “prover”, convinces one or more “verifiers” that it has knowledge of a witness attesting to the truth of a given statement. The correct prover and verifier algorithms are denoted as  $P$  and  $V$  respectively. Arbitrary, and potentially malicious, prover and verifier algorithms are denoted as  $P^*$  and  $V^*$ .

The prover in the protocol must convince the verifier that it has a witness to the specified statement being true. If the statement is false, an arbitrary, computationally

unbounded prover algorithm  $P^*$ , must be incapable of convincing  $V$  that the statement is true. When an assumption of computational boundedness on  $P^*$  is required to prove the soundness of the protocol, the resulting protocol is referred to as a zero-knowledge argument of knowledge. At the same time, the zero-knowledge property of a ZKPoK requires that an arbitrary verifier algorithm  $V^*$  learn nothing from the protocol except that the prover possesses the claimed witness, and whatever information can be inferred from that.

A key concept when defining the zero-knowledge properties of zero-knowledge proofs and arguments is *simulation*. Now, a protocol  $(V, P)$  is said to be “simulatable” if an arbitrary verifier algorithm  $V^*$  can simulate, without any information from  $P^*$ , an interaction which is indistinguishable from a real interaction between  $P$  and  $V^*$  on the same common input. This implies that any information that  $V^*$  can learn using any strategy in this protocol can be easily computed using the common input  $s$  and the knowledge that the given statement is true on input  $s$ .

A central component in the formalization of simulatability is the *transcript* of the execution of a protocol. The transcript is simply a tuple of the messages sent between  $V^*$  and  $P$ . This is then used to create the *view* of the verifier  $V^*$ . Intuitively, the view, denoted  $\mathbf{view}_{P,V^*}(s)$ , is simply the information that  $V^*$  “sees” when the protocol is executed. This consists of, the transcript, and the common input  $s$  as well as any private inputs given to  $V^*$  and any random coin flips performed by  $V^*$ . All ZKPs need at least one of the participant algorithms to be probabilistic, so  $\mathbf{view}_{P,V^*}(s)$  is a random variable.

## 2.5 Secure Multiparty Computation (MPC)

A *secure multiparty computation* (MPC) protocol is a cryptographic protocol through which two or more mutually distrusting parties jointly compute some function over their private inputs while disclosing nothing to any curious parties beyond what that curious party can deduce from its own private input, the shared public output, and any private knowledge it possesses [63]. In an MPC protocol taking place between  $\ell$  parties, we denote the parties as  $P_b$  where  $0 \leq b < \ell$ . In addition to the functionality being computed, MPC protocols are typically parameterized by (i) the number “ $\ell$ ” of parties to the computation, and (ii) the size “ $t + 1$ ” of coalition needed to violate the

privacy of non-coalition members' inputs.

The concepts of simulatability, transcripts, and views, introduced for the ZKPs in [Section 2.4](#), directly translate into MPC protocols. The privacy of a  $t$ -private MPC protocol is based upon simulatability. However, the simulation can be performed by an attacker who controls, and sees the views of, a coalition of multiple parties in the protocol, as long as the coalition size is no larger than the maximum privacy protecting coalition size  $t$ . The concept of a transcript is similarly expanded in MPC, namely for each pair of parties there is a transcript of messages sent between those two parties. A simulated party's view then includes all transcripts between that party and any other parties in the protocol.

### **MPC adversary models**

In MPC, there are two primary types of adversaries to consider. These are semi-honest adversaries and malicious adversaries. A semi-honest party, also known as an "honest-but-curious" party, is simply a party in an MPC protocol which must follow the protocol, but may attempt to undermine the privacy guarantees of the protocol by observing intermediate results in the computation. In contrast, a malicious party is not required to follow the protocol in any way. This type of adversary can send arbitrary messages in order to extract private information that it should not have access to [\[32\]](#).

### **Secure $(2 + 1)$ -party computation**

One special case of MPC is so-called *secure  $(2 + 1)$ -party computation* (also known as *server-aided 2-party computation*), wherein two mutually distrusting parties enlist the help of a semi-honest third party (who provides no input and receives no output) to assist in the computation. The computation performed by this third party is independent of the actual data being processed and can be computed ahead of time in a precomputation step. Such protocols have received considerable attention due to their superior performance relative to "pure" 2-party protocols or  $\ell$ -party protocols secure against stronger attackers. In addition,  $(2 + 1)$ -party protocols are generally preferable over three-party protocols because the role of the assisting server is data independent. Therefore, it can be replaced with a two-party precomputation protocol

that computes the same values, easily converting a  $(2+1)$ -party computation protocol into an equivalent two-party protocol. Since the role of the assisting party is replaced by a precomputation phase that is completely agnostic to the parties' inputs, the online portion of the two-party protocol is at least as fast as the corresponding  $(2 + 1)$ -party protocol.

### **MPC-in-the-head**

*MPC-in-the-head* [16, 27, 39, 42] is a conceptual framework for constructing zero-knowledge proofs of knowledge (ZKPoKs) from MPC building blocks. ZKPoKs based on MPC-in-the-head leverage the dual observations that (i) MPC protocols are *simulatable*—even from the perspective of an “insider” (i.e., an attacker who controls a size- $t$  subset of the parties to the computation, where  $t$  is the maximum number of parties that can collude without compromising the privacy of non-colluding parties' inputs)—and (ii) dishonest behaviour by a coalition in an MPC protocol execution necessarily yields incontrovertible evidence in the coalition members' joint view.

Consider an MPC protocol  $\Pi$  for the witness-checking procedure of some NP relation  $R$ . In light of the preceding two observations, one can transform  $\Pi$  into a ZKPoK for  $R$  as follows: Given a (public) instance  $I$  and (private) witness  $w$  such that  $R(I, w) = 1$ , the *prover* secret shares  $w$  among  $\ell$  imaginary computation parties, and then it *simulates* a complete run of  $\Pi$  among these imaginary parties, up to and including the point where the functionality attests to the validity of  $w$  by outputting 1. The prover *commits* (say, via Merkle tree) to the simulated view of each imaginary party, presenting the resulting commitments to the verifier.

If the prover is attempting to cheat, then the correctness property of the MPC protocol ensures that incontrovertible evidence must exist in at least one coalition's view. Thus, the verifier attempts to uncover such evidence by challenging the prover to reveal the view of a random size- $t$  coalition of parties. The probability of detecting cheating varies based on the collusion threshold  $t$  and number of imaginary parties  $\ell$ . To amplify the probability of detecting cheating, the prover and verifier apply the cut-and-choose technique [48], engaging in multiple parallel instances of the procedure of simulating the protocol, committing to simulated views, and revealing a

randomly chosen size- $t$  coalition’s view to the verifier, each one operating on a “fresh” simulation. If a single simulation has a probability  $p$  of detecting catching prover and  $n$  simulations are performed for the cut-and-choose technique, the resulting probability of catching a cheating prover increases to  $1 - (1 - p)^n$ .

### 2.5.1 Secret share-based MPC

Secret share-based MPC is an approach to MPC that relies upon using private inputs stored as additive or XOR secret shares [33]. Because additive secret shares are based upon modular addition, they are especially amenable to linear operations. Specifically, given additive secret shares  $[x]_i$  and  $[y]_i$  along with public constants  $a$  and  $b$ , party  $i$  can locally compute  $[a \cdot x + b \cdot y]_i = a \cdot [x]_i + b \cdot [y]_i$ . While multiplication of two secret shared values is more difficult, Beaver Triples, as discussed in [Section 2.5.3](#), allow for two secret shared numbers to be multiplied using precomputation and one online round of communication. A primary advantage of this type of protocol is that expensive operations using oblivious transfer (OT) and arithmetic homomorphic encryption (AHE) can be completely moved into the preprocessing phase of the protocol or, sometimes, replaced through the use of additional parties in the computation as described in [Section 2.5](#). While OT and AHE along with their use in MPC protocols, such as the precomputation of Beaver triples [5, 15, 28], are important, the details of these protocols are not relevant to the work discussed in this thesis. Moving these computationally costly operations into precomputation comes at the cost of higher round complexities. At the start of a secret share-based MPC protocol, all private inputs are secret shared according to the secret sharing protocol being used. As described in [Section 2.3.1](#), additive shares are linearly homomorphic, allowing additively secret shared values to be multiplied by non-secret shared constants and added together without any communication. Similarly, XOR shares, as  $n$ -tuples of additive shares over  $\mathbb{Z}_2$ , are also linearly homomorphic with bitwise exclusive-OR as “addition” and bitwise logical-AND as “multiplication”. Multiplication of two secret shared values is commonly performed using Beaver multiplication triples [6].

### 2.5.2 Garbled circuit-based MPC

Another method for performing MPC is to use Yao’s Garbled circuit technique [64]. The techniques and developments presented in this thesis do not use garbled circuits, but we do contrast some of our constructions against garbled circuit-based techniques. Therefore, this section only presents a high-level explanation of garbled circuits.

This type of protocol is performed by two parties, a *garbler* and an *evaluator*. The garbler prepares a garbled circuit made up of *garbled gates*. These are encrypted forms of the truth tables for the corresponding Boolean gates. Each bit of input, from either party, has a key corresponding to a value of one on that bit and a key corresponding to a zero bit. An output  $\gamma_{g,e}$  corresponding to an input bit  $g$  from the garbler and  $e$  from the evaluator, is then encrypted by the garbler using a combination of keys  $W_G^g$  and  $W_E^e$ . Thus, the output can only be decrypted by a party who knows both of those keys. The garbler randomizes the order of these ciphertext outputs and sends them to the evaluator along with the keys corresponding to the garbler’s private inputs. Thus, without revealing the garbler’s bits, the evaluator cannot decrypt outputs which don’t correspond to the garbler’s inputs. In order to get the keys for the evaluators’ input bits, OT is used to retrieve only  $W_E^e$  from the garbler without revealing  $e$ . The details of how the various versions of OT work is not important for this explanation. Then the evaluator has the keys to decrypt the output ciphertext corresponding to the true inputs  $g$  and  $e$  and no other output ciphertext. If this output is to be used as an input to another garbled gate in the circuit, the encrypted output is another key corresponding to an input for those later gates. Otherwise, it can be the output value of the gate.

One key strength of garbled circuits is that their performance is not related to the linearity of the computation being performed. This is in contrast to secret share-based MPC protocols which are extremely efficient at linear computations, but much less efficient when computing non-linear functions. As a result, garbled circuits are generally the method most suited to computations that are largely non-linear whereas primarily linear computations are more efficient in secret share-based MPC protocols.

Garbled circuits also have an advantage in round complexity since they only

require a small constant number of rounds of communication for evaluating any circuit. However, each gate in the circuit requires all of the encrypted ciphertext outputs to be sent by the garbler along with the garbler’s input keys. This results in a significant communication cost. The further communication and computation cost of using OT to retrieve the evaluator’s input keys significantly slows the protocols as does the need to perform multiple decryptions for each gate in the circuit. The basic form of garbled circuits has been extended with various optimizations such as point-and-permute [8] and free XOR [44] which respectively remove the extra decryptions required to find the output ciphertext and allow for XOR to be performed without needing any encryption operations. The half-gates optimization, proposed by Zahur, Rosulek, and Evans, is another feature that has been added to garbled circuits [66]. Half-gates allow garbled AND gates to be computed using only two ciphertexts rather than the four ciphertexts required in the original version of garbled circuits. Garbled circuit computations have also reduced the number of OTs required by employing the OT extension technique. An OT extension is simply a method for taking a small number of standard OTs and then using fast symmetric cryptography to stretch those initial OTs into a large number of OTs, without the expensive computations required for standard OTs [7, 38]. OT extension can be further optimized for garbled circuits through the use of correlated OT (C-OT) [4]. C-OT relies upon the fact that when using the free-XOR and point-and-permute technique, the two keys,  $k_0$  and  $k_1$ , which the evaluator could choose for a given OT are always correlated. In particular,  $k_0$  is chosen at random and  $k_1 = k_0 \oplus \Delta$ , where  $\Delta$  is a randomly selected bitstring where the least significant bit is always set. C-OT exploits this relationship to generate two keys with the required correlation and then obviously transfer one to the evaluator more efficiently than is possible using normal OT extension. Since the keys in C-OT are generated as part of the OT extension, this optimization does have the side effect of forcing the garbler to garble the circuit after performing the OT extension. Despite these optimizations, garbled circuits still have a high communication and computation cost due to key sizes, oblivious transfers, and decryption operations. To address these downsides, other MPC techniques have been proposed.



### 2.5.3 Beaver multiplication triples

Beaver multiplication triples [6] enable the efficient multiplication of  $(2, 2)$ -additively shared secrets. Each triple comprises three shares  $([X], [Y], [Z])$ , where  $X$  and  $Y$  are uniform random scalars and

$$Z := [X]_0 \cdot [Y]_1 + [X]_1 \cdot [Y]_0.$$

The shareholders typically precompute Beaver multiplication triples using either additively homomorphic encryption [52] or oblivious transfer [55] during a (rather costly) precomputation phase; alternatively, in the case of  $(2 + 1)$ -party computation, well-formed triples are provided to the shareholders for “free” by the semi-honest third party [24].

Given a pair of shares  $[x]$  and  $[y]$  and a Beaver triple  $([X], [Y], [Z])$ , each shareholder  $b$  sends

$$([x + X]_b, [y + Y]_b) := ([x]_b + [X]_b, [y]_b + [Y]_b)$$

to its peer, and then it outputs

$$[z]_b := [x]_b \cdot ([y]_b + [y + Y]_{1-b}) - [Y]_b \cdot [x + X]_{1-b} + [Z]_b$$

so that

$$\begin{aligned} [z]_0 + [z]_1 &= ([x]_0 \cdot ([y]_0 + [y + Y]_1) \\ &\quad - [Y]_0 \cdot [x + X]_1 + [Z]_0) \\ &\quad + ([x]_1 \cdot ([y]_1 + [y + Y]_0) \\ &\quad - [Y]_1 \cdot [x + X]_0 + [Z]_1) \\ &= ([x]_0 \cdot ([y]_0 + ([y]_1 + \cancel{[Y]_1})) \\ &\quad - [Y]_0 \cdot (\cancel{[x]_1} + \cancel{[X]_1}) + \cancel{[Z]_0}) \\ &\quad + (\cancel{[x]_1} \cdot ([y]_1 + ([y]_0 + \cancel{[Y]_0})) \\ &\quad - \cancel{[Y]_1} \cdot (\cancel{[x]_0} + \cancel{[X]_0}) + \cancel{[Z]_1}) \\ &= [x]_0 \cdot [y]_0 + [x]_0 \cdot [y]_1 + [x]_1 \cdot [y]_1 + [x]_1 \cdot [y]_0 \\ &= x \cdot y. \end{aligned}$$

Beaver triples are ephemeral, each enabling just a single multiplication. Each multiplication requires a round of communication. When multiple multiplications

need to be performed. The same round of communication can exchange the values, blinded using the Beaver triples, for multiple multiplications in parallel. The resulting number of rounds depends upon the multiplicative depth of the computation. For example, consider secret shared numbers  $[v]$ ,  $[w]$ ,  $[x]$ ,  $[y]$ ,  $[z]$ . In order to compute shares of  $x \cdot y + (w \cdot v) \cdot z$ , the parties first use a round of communication and two Beaver triples to compute  $[x \cdot y]$  and  $[w \cdot v]$  in parallel. Another round of communication and a third Beaver triple can then be used to compute  $[(w \cdot v) \cdot z]$  from  $[z]$  and  $[w \cdot v]$ . The final sum can then be computed locally by the parties.

Beaver multiplication is also agnostic to whether  $x$  and  $y$  represent “actual” integers or fixed-point numbers. Of course, in the latter case, multiplication will increase the fractional bits in  $z$  relative to  $x$  and  $y$ , possibly necessitating a subsequent fractional bit reduction.

### **Du-Atallah multiplication**

While Beaver triples are designed for 2-party MPC with precomputation, the same concept can be applied to  $(2 + 1)$ -party protocols as well. This variant is referred to as Du-Atallah multiplication [24].

In this setting,  $P_2$ , the assisting party, selects  $[X]$  and  $[Y]$  at random. It then computes  $[Z]$  using the shares of  $X$  and  $Y$ .  $P_2$  then sends  $([X]_b, [Y]_b, [Z]_b)$  to party  $P_b$ , for  $b \in \{0, 1\}$ .

$P_0$  and  $P_1$  then proceed to use the multiplication triple in exactly the same way they would use a precomputed triple in a 2-party setting.

### **Plaintext Du-Atallah multiplication**

In an MPC multiplication of  $x$  and  $y$  where  $P_0$  holds  $x$  as a plaintext and  $P_1$  holds  $y$  as a plaintext, a variant of Du-Atallah can be used. The same idea can also be applied to Beaver triples in order to produce a Beaver triple variant for multiplying plaintext values with analagous performance improvements.

The modified multiplication triple consists of  $(X, Y, [Z])$  where  $Z = X \cdot Y$ . In this protocol,  $P_0$  receives  $X$  and  $[Z]_0$ . Similarly,  $P_1$  receives  $Y$  and  $[Z]_1$ .

$P_0$  sends  $x + X$  to  $P_1$ , and  $P_1$  sends  $y + Y$  to  $P_0$ . The resulting shares of  $z = x \cdot y$  are then,

$$\begin{aligned} [z]_0 &= [Z]_0 + x \cdot (y + Y) \\ [z]_1 &= [Z]_1 - Y \cdot (x + X) \end{aligned}$$

We can confirm the correctness of these results, by computing  $z$ ,

$$\begin{aligned} z &= [z]_0 + [z]_1 \\ &= ([Z]_0 + x \cdot (y + Y)) + ([Z]_1 - Y \cdot (x + X)) \\ &= ([Z]_0 + [Z]_1) + x \cdot y + x \cdot Y - Y \cdot x - Y \cdot X \\ &= Z + x \cdot y - Y \cdot X \\ &= Y \cdot X + x \cdot y - Y \cdot X \\ &= x \cdot y \end{aligned}$$

Compared to standard Du-Atallah, this protocol reduces the number of server computed values sent to each party and the online communication cost. The reduction in precomputed or server computed values comes from the fact that each party only receives two numbers,  $X$  and  $[Z]_0$  for  $P_0$  and  $Y$  and  $[Z]_1$  for  $P_1$ , as opposed to standard Du-Atallah in which each party receives one share from each of  $[Z]$ ,  $[X]$ , and  $[Y]$ . Since each party only sends one blinded value,  $x + X$  for  $P_0$  and  $y + Y$  for  $P_1$ , rather than two the communication cost of Du-Atallah on plaintexts is half that of standard Du-Atallah.

### **D-ary MPC Multiplication**

As can be seen in the earlier example, when computing the product of  $D$  elements  $a_1 \cdot a_2 \cdots a_D$ , known as a  $D$ -ary multiplication, standard Beaver triples require  $D - 1$  rounds of communication. In ABY2.0, Patra et al. show that the ideas underlying MPC multiplication using Beaver triples and Du-Atallah multiplication can be extended, using additional precomputation, to allow for  $D$ -ary MPC multiplication to be performed using a single round of communication [54].

Consider the following derivation for MPC 3-ary multiplication of  $[x]$ ,  $[y]$ , and  $[z]$ . The precomputation chooses three random values  $X$ ,  $Y$ , and  $Z$  to be the *blinding*

factors for  $x$ ,  $y$ , and  $z$  respectively. In the one round of communication, the parties exchange shares of  $\bar{x} = x + X$ ,  $\bar{y} = y + Y$ , and  $\bar{z} = z + Z$ . From there, the necessary precomputed values can be found by expanding the desired product.

$$\begin{aligned} [xyz] &= (\bar{x} - [X]) \cdot (\bar{y} - [Y]) \cdot (\bar{z} - [Z]) \\ &= \bar{x} \cdot \bar{y} \cdot \bar{z} - \bar{x} \cdot \bar{y} \cdot [Z] - \bar{x} \cdot \bar{z} \cdot [Y] - \bar{y} \cdot \bar{z} \cdot [X] + \bar{x} \cdot [YZ] + \bar{y} \cdot [XZ] + \bar{z} \cdot [XY] - [XYZ] \end{aligned}$$

Thus, the precomputation needs to compute  $[XY]$ ,  $[XZ]$ ,  $[YZ]$ , and  $[XYZ]$  in addition to  $[X]$ ,  $[Y]$ , and  $[Z]$ . Party  $P_b$  receives shares  $[X]_b$ ,  $[Y]_b$ ,  $[Z]_b$ ,  $[XY]_b$ ,  $[XZ]_b$ ,  $[YZ]_b$ , and  $[XYZ]_b$ . This allows it to locally compute  $[xyz]_b = b \cdot \bar{x} \cdot \bar{y} \cdot \bar{z} - \bar{x} \cdot \bar{y} \cdot [Z]_b - \bar{x} \cdot \bar{z} \cdot [Y]_b - \bar{y} \cdot \bar{z} \cdot [X]_b + \bar{x} \cdot [YZ]_b + \bar{y} \cdot [XZ]_b + \bar{z} \cdot [XY]_b - [XYZ]_b$ .

This same process can be performed for any choice of  $D$  in order to produce a single-round  $D$ -ary MPC multiplication with the amount of precomputed terms increasing as  $D$  increases.

#### 2.5.4 The ABY framework

In order to achieve a balance of the advantages and disadvantages of different MPC methods, mixed-protocol frameworks have been proposed. These frameworks combine multiple methods with the goal of emphasizing the strengths of each method while minimizing the impact of the weaknesses of each method. The ABY framework [22] is one of the most popular mixed-protocol frameworks. This framework combines arithmetic secret share, XOR secret share, and garbled circuit-based two-party MPC protocols. It does this using two party protocols for converting between these different representations of the secret values.

Mixed-protocol frameworks were further developed by Patra et al.'s ABY2.0 [53]. This framework used increased preprocessing and modified secret shares in order to reduce the required online cost of its protocols. In particular, ABY2.0 makes heavy use of a version of the  $D$ -ary multiplication discussed in Section 2.5.3 which improves online costs, at the cost of increased preprocessing.

ABY has also been extended to work for 3-party MPC with the ABY<sup>3</sup> framework by Mohassel and Rindal [50]. In ABY<sup>3</sup>, a secret  $v$  is additively shared into three pieces  $[v]_0$ ,  $[v]_1$ , and  $[v]_2$  where  $v = [v]_0 + [v]_1 + [v]_2$ . Each party is given two of these shares, to form a  $(2, 3)$ -threshold additive secret sharing. Additionally, at an

increased communication cost, ABY<sup>3</sup> can provide security against actively malicious participants in addition to the passive adversaries protected against by ABY and ABY2.0.

## 2.6 Selection vector

A *selection vector* is a vector in which one element is 1 and all others are 0. We refer to the length- $N$  selection vector having its 1 in position  $i \in [0 \dots N)$  as the  $i$ th *selection vector of length  $N$* , and we denote this vector by  $\vec{e}_i$ . This is also referred to as the *one-hot vector* encoding the number  $i$ .

Selection vectors have a variety of applications in information-theoretic secure PIR and PIR-writing as described in [Section 2.8](#). However, the scalability of these techniques are limited, because the size of the vector grows linearly with  $N$ .

## 2.7 Point functions

A *binary point function* is a “functional” representation of a selection vector; that is, a Boolean-valued function that has a selection vector as its truth table.

**Definition 2.** The  $i$ th *binary point function* on  $\mathbb{Z}_N$  is the function  $P_i: \mathbb{Z}_N \rightarrow \{0, 1\}$  for which

$$\vec{e}_i := (P_i(0), P_i(1), \dots, P_i(N-1))$$

is the  $i$ th selection vector of length  $N$ .

*Generalized point functions* expand on binary point functions by functionally representing a selection vector which has been scaled by some value. Thus, the truth table of a generalized point function is zero at all indices except for one, which contains the point function’s specified output value  $y$  [\[29\]](#).

**Definition 3.** The *generalized point function*  $P_{x,y}: \mathbb{Z}_N \rightarrow \mathbb{G}$  is the function defined by

$$P_{x,y}(i) := \begin{cases} y & \text{if } i = x, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

Where  $\mathbb{G}$  is a group, and  $y \in \mathbb{G}$ .

Both binary and generalized point functions are used in the following chapters.

### 2.7.1 Distributed point functions

Intuitively, a *distributed point function* (DPF) is a compact, secret-shared representation of a point function [29]. We wrote the following formal definition specialized for the case of  $(2, 2)$ -DPFs with arbitrary domain and range.

**Definition 4.** A  $(2, 2)$ -distributed point function, or  $(2, 2)$ -DPF, is a pair of PPT algorithms  $(\text{Gen}, \text{Eval})$  defining an infinite family of secret-shared representations of generalized point functions; that is, given (i) a security parameter  $\lambda \in \mathbb{N}$ , (ii) a domain  $D$  and range  $R$ , where there exists an additive identity  $\mathbf{0} \in R$ , and (iii) a distinguished point  $(x, y) \in D \times R$ , Then, we have

1. **Correctness:** If  $(\llbracket(x, y)\rrbracket_0, \llbracket(x, y)\rrbracket_1) \leftarrow \text{Gen}(1^\lambda, D, R; x, y)$ , then, for all  $i \in D$ ,

$$\text{Eval}(\llbracket(x, y)\rrbracket_0, i) + \text{Eval}(\llbracket(x, y)\rrbracket_1, i) := \begin{cases} y & \text{if } i = x, \text{ and} \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

2. **Simulatability:** There exists a PPT simulator  $\mathcal{S}$  such that, for any given domain  $D$ , range  $R$ , distinguished point  $(x, y) \in D \times R$ , and bit  $b \in \{0, 1\}$ , the distribution ensembles

$$\{\mathcal{S}(1^\lambda, D, R; b)\}_{\lambda \in \mathbb{N}}$$

and

$$\{\llbracket(x, y)\rrbracket_b \mid (\llbracket(x, y)\rrbracket_0, \llbracket(x, y)\rrbracket_1) \leftarrow \text{Gen}(1^\lambda, D, R; x, y)\}_{\lambda \in \mathbb{N}}$$

are computationally indistinguishable. In the case of the Boyle–Gilboa–Ishai DPF construction discussed in [Section 3.2](#), each key is made up entirely of random and pseudorandom numbers. Therefore  $\mathcal{S}(1^\lambda, D, R; b)$  simply needs to produce an output with the same format as a real DPF key, but all entries are sampled uniformly at random from the appropriate ring or group. As a result, the two distribution ensembles are computationally indistinguishable.

The  $\llbracket(x, y)\rrbracket_b$  output by Gen are called  $(2, 2)$ -DPF keys; the  $x$ -coordinate of the distinguished point is the *distinguished input*.

In the special case where the domain is  $D = \mathbb{Z}_N$  and the output is a 1-bit value (i.e.,  $R = \mathbb{Z}_2$ ), the DPF corresponds to a binary point function. The generation function for a binary DPF is denoted as  $\text{Gen}(1^\lambda, N; x)$  with output  $(\llbracket\vec{e}_x\rrbracket_0, \llbracket\vec{e}_x\rrbracket_1)$ . This is equivalent to the general DPF generation function  $\text{Gen}(1^\lambda, \mathbb{Z}_N, \mathbb{Z}_2; x, 1)$ . That is, the input is a non-negative number less than the upper bound  $N \in \mathbb{N}$ , and, on the distinguished input  $x \in \mathbb{Z}_N$ , the evaluation function outputs XOR shares of  $y = 0$  or  $y = 1$ .

The next chapter introduces the Boyle–Gilboa–Ishai construction for  $(2, 2)$ -DPFs [14], and its structural properties. This design is the most efficient DPF construction currently available, producing keys of size  $n(\lambda + 2)$  bits where  $n$  is the input size in bits and  $\lambda$  is the security parameter.

## 2.8 Private information retrieval (PIR)

PIR is a cryptographic primitive which enables a reader to retrieve data from an untrusted database server or servers without revealing what data they fetched [18]. A closely related idea is PIR-writing, where data is obviously written into a database held by untrusted servers without revealing what data was written or where it was written.

### 2.8.1 Private information retrieval (PIR) from selection vectors

As their name hints, selection vectors are useful for selecting items from a list. For example, consider a database  $\vec{P}$ . By encoding this database as a vector and taking an inner product with  $\vec{e}_j$ , we find that  $\langle \vec{e}_j, \vec{P} \rangle = P_j$ , where  $P_j$  is the  $j$ th element of  $\vec{P}$  [18]. In order to keep the data being written private from the servers holding the database, the selection vector  $\vec{e}_j$  can be secret shared into multiple vectors  $[\vec{e}_j]_0, [\vec{e}_j]_1, \dots, [\vec{e}_j]_{\ell-1}$  such that  $\vec{e}_j = [\vec{e}_j]_0 + [\vec{e}_j]_1 + \dots + [\vec{e}_j]_{\ell-1}$ . The two server version of this protocol is shown in Figure 2.1 where the client is denoted as  $C$  and the two servers are  $S_0$  and  $S_1$ .

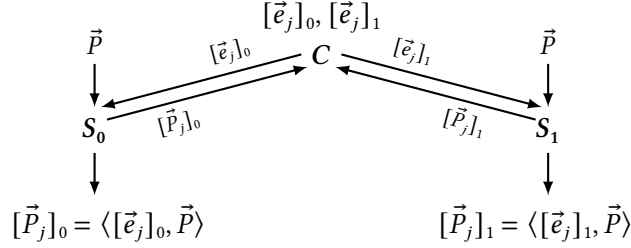


Figure 2.1: A two server information theoretic secure PIR protocol using selection vectors.

Since the selection vector  $\vec{e}_j$  contains exactly one entry with the value one, and all other entries have the value zero, we know that  $[\vec{P}_j]_0$  and  $[\vec{P}_j]_1$  in Figure 2.1 sum to zero at all indices except  $j$ , where they sum to one. Therefore, the inner products  $[\vec{P}_j]_0 = \langle [\vec{e}_j]_0, \vec{P} \rangle$  and  $[\vec{P}_j]_1 = \langle [\vec{e}_j]_1, \vec{P} \rangle$  sum to  $[\vec{P}_j]_0 + [\vec{P}_j]_1 = \langle [\vec{e}_j]_0, \vec{P} \rangle + \langle [\vec{e}_j]_1, \vec{P} \rangle = \langle \vec{e}_j, \vec{P} \rangle = \vec{P}_j$ . Thus, the result returned by the protocol in Figure 2.1 is  $[\vec{P}_j]_0 + [\vec{P}_j]_1 = \vec{P}_j$ , the  $j$ th element in  $\vec{P}$ .

Additionally, if  $\vec{e}_j$  is XOR shared, rather than being additively shared, into  $\vec{e}_j = (\vec{e}_j)_0 \oplus (\vec{e}_j)_1 \oplus \dots \oplus (\vec{e}_j)_{\ell-1}$  then the same protocol can be performed except using XOR instead of addition. This includes the inner product of  $\langle (\vec{e}_j)_i, \vec{P} \rangle$  which becomes  $\bigoplus_{k=0}^{N-1} (\vec{e}_j)_{i,k} \cdot \vec{P}_k$  when using XOR. The same logic used in the additive sharing setting continues to apply, so, when using XOR shares, the result becomes  $(\vec{P}_j)_0 \oplus (\vec{P}_j)_1 = \vec{P}_j$ .

Additionally, since each share of the selection vector is random, the sharing is information theoretically secure. As a result, the overall protocol has information theoretic security.

### PIR example

Consider a toy example of a PIR system in which there are two servers  $S_0$  and  $S_1$ . Let the database held by these servers be the vector containing  $N = 4$  bitstrings

$$\vec{D} = \begin{pmatrix} 101010 & 001100 & 000111 & 110011 \end{pmatrix}$$

Suppose that a client wants to retrieve the value at index  $j = 0$ , which we know to be 101010. The client can then generate XOR shares  $[\vec{e}_j]_0$  and  $[\vec{e}_j]_1$  of  $\vec{e}_j$ , for example



we use

$$\begin{aligned} [\vec{e}_j]_0 &= \begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix} \\ [\vec{e}_j]_1 &= \begin{pmatrix} 0 & 0 & 1 & 1 \end{pmatrix} \end{aligned}$$

The client proceeds to send  $[\vec{e}_j]_0$  to  $S_0$  and  $[\vec{e}_j]_1$  to  $S_1$ .  $S_0$  then computes the inner product of  $[\vec{e}_j]_0$  and  $\vec{D}$ , which gives us

$$\begin{aligned} \langle \vec{P}_j \rangle_0 &= 101010 \cdot 1 \oplus 001100 \cdot 0 \oplus 000111 \cdot 1 \oplus 110011 \cdot 1 \\ &= 101010 \oplus 000111 \oplus 110011 \\ &= 011110 \end{aligned}$$

Similarly,  $S_1$  computes the inner product of  $[\vec{e}_j]_1$  and  $\vec{D}$ , which gives us

$$\begin{aligned} \langle \vec{P}_j \rangle_1 &= 101010 \cdot 0 \oplus 001100 \cdot 0 \oplus 000111 \cdot 1 \oplus 110011 \cdot 1 \\ &= 000111 \oplus 110011 \\ &= 110100 \end{aligned}$$

When the client receives  $\langle \vec{P}_j \rangle_0$  and  $\langle \vec{P}_j \rangle_1$  from the servers, it can compute  $\langle \vec{P}_j \rangle_0 \oplus \langle \vec{P}_j \rangle_1 = 011110 \oplus 110100 = 101010$ , which is the bitstring stored at index  $j$  in the database  $\vec{D}$ .

### 2.8.2 DPF-based PIR

One of the earliest proposed applications for DPFs is two-server PIR [29]. As discussed in Section 2.8.1, PIR is the problem of retrieving an entry from a database without revealing which entry was accessed.

The selection vector-based PIR protocol shown in Figure 2.1 can easily be converted to use DPFs for PIR in place of selection vectors. To do this, the generation of the vectors  $[\vec{e}_j]_0$  and  $[\vec{e}_j]_1$  is removed. Instead, a DPF key pair  $(\llbracket \vec{e}_j \rrbracket_0, \llbracket \vec{e}_j \rrbracket_1) \leftarrow \text{Gen}(1^\lambda, N; j)$  is generated. DPF key  $\llbracket \vec{e}_j \rrbracket_i$  is then sent to server  $S_i$  in place of the vector  $[\vec{e}_j]_i$ . When the server receives the DPF key, it performs an evaluation of the DPF at all  $N$  points within its domain producing the vector,  $[\vec{e}_j]_i$  such that  $[\vec{e}_j]_0 \oplus [\vec{e}_j]_1 = \vec{e}_j$  exactly like the query vectors from Figure 2.1. The server then calculates the inner product  $[\vec{P}_j]_i = \langle [\vec{e}_j]_i, \vec{P} \rangle$ , as in the selection vector-based PIR

protocol. The client then receives the response shares which again have the property that  $[\vec{P}_j]_0 \oplus [\vec{P}_j]_1 = \vec{P}_j$ , producing the desired output.

The process of evaluating a DPF at all  $N$  points within its domain is referred to as a *full-domain evaluation*. The naive method for performing a full domain evaluation is to simply iterate through all  $N$  values in the domain, and evaluate the DPF at each point. In the Boyle–Gilboa–Ishai DPF construction, a single evaluation has time complexity  $O(n)$ , where  $n$  is the previously defined height of a point function tree. Thus, the naive method requires  $O(nN)$  time. As will be discussed in more detail in [Section 3.2](#), the Boyle–Gilboa–Ishai DPF construction is based on the tree structure of a point function. This allows for several optimizations in the full-domain evaluation which were proposed in [\[14\]](#). The first optimization is to notice that each  $k$ -bit leaf only needs to be computed once to get all  $k$  inputs that fall within that leaf this reduces the time complexity to  $O(n2^n)$ . The second optimization takes this idea even further and only calculates each node in the point function tree once. This reduces the time complexity to  $O(2^n)$ , since there are  $2^n - 1$  nodes in the point function tree.

Using a DPF in place of a secret shared vector means that this protocol is no longer information-theoretically secure, as DPFs rely on the mild computational assumption of the existence of a one-way function rather than on information-theoretic assumptions. However, the use of DPFs also significantly reduces the communication cost from  $N = 2^n$  bits sent to each server down to  $n(\lambda + 2)$  bits sent to each server, when using the most space efficient logarithmic-size DPF construction from Boyle et al. [\[14\]](#).

### 2.8.3 DPFs for private writing

Another application proposed for DPFs is PIR-writing, where a client privately writes to a secret shared database [\[29\]](#). In this application, the domain is  $\mathbb{Z}_N$ , and the database  $D$  is secret shared into  $\langle D \rangle_0$  and  $\langle D \rangle_1$ . The share  $\langle D \rangle_b$  is held by server  $S_b$ , for  $b \in \{0, 1\}$ . A client wanting to write an value  $v \in \mathbb{Z}_2^m$ , where  $m$  is the bitlength of entries in  $D$ , to location  $i$  can then generate a DPF key pair  $(\llbracket (i, v) \rrbracket_0, \llbracket (i, v) \rrbracket_1) \leftarrow \text{Gen}(1^\lambda, \mathbb{Z}_N, \mathbb{Z}_2^m; i, v)$ . In this case, the DPF outputs are the same size as the entries in the database. The key  $\llbracket (i, v) \rrbracket_b$  is then sent to server

$S_b$ . Upon receiving the key, server  $S_b$  evaluates  $\llbracket(i, v)\rrbracket_b$  across its whole domain expanding it into the vector  $\langle d \rangle_b$ , where  $\langle d \rangle_0 \oplus \langle d \rangle_1 = d$  the vector with  $v$  at location  $i$  and zero at all other locations, as required for the output of a DPF. Server  $S_b$  then updates its database share to be  $\langle D' \rangle_b = \langle D \rangle_b \oplus \langle d \rangle_b$ . When the plaintext of the database is reconstructed, the result is  $D' = D \oplus d$  which will write the value  $v$  to index  $i$  as required.

To prove the correctness of the write, consider an arbitrarily selected index  $k \in \mathbb{Z}_N$ . We will denote the  $k$ th entry in  $\langle d \rangle_b$  as  $\langle d_k \rangle_b$ . Similarly, the  $k$ th entry in  $\langle D \rangle_b$  will be denoted as  $\langle D_k \rangle_b$ . Obviously, the plaintexts corresponding to these shares are  $d_k = \langle d_k \rangle_0 \oplus \langle d_k \rangle_1$  and  $D_k = \langle D_k \rangle_0 \oplus \langle D_k \rangle_1$ , the existing  $k$ th value in  $D$ . Now, there are two cases to consider.

**$k = i$ :** In this case, the definition of a DPF requires  $v = d_k$ . When the write is performed, the  $k$ th element in  $\langle D' \rangle_b$  is  $\langle d_k \rangle_b \oplus \langle D_k \rangle_b$ . Reconstructing the  $k$ th element of  $D'$  gives us

$$\begin{aligned} & (\langle d_k \rangle_0 \oplus \langle D_k \rangle_0) \oplus (\langle d_k \rangle_1 \oplus \langle D_k \rangle_1) \\ &= \langle d_k \rangle_0 \oplus \langle d_k \rangle_1 \oplus \langle D_k \rangle_0 \oplus \langle D_k \rangle_1 = d_k \oplus D_k \\ &= v \oplus D_k \end{aligned}$$

If no prior value has been written at index  $k$ , then  $D_k = \mathbf{0}$ , and the plaintext value is  $v$ . Similarly, if a value  $u$  has already been written to index  $k$ , we have  $D_k = u$ , so the plaintext entry is  $v \oplus u$ . This is exactly what we expect for the index being written to, so we can conclude that the message has been written at the desired index  $i$ .

**$k \neq i$ :** Here, the definition of a DPF tells us that  $d_k = \mathbf{0}$ . Reconstructing the  $k$ th element of  $D'$  gives us

$$\begin{aligned} & (\langle d_k \rangle_0 \oplus \langle D_k \rangle_0) \oplus (\langle d_k \rangle_1 \oplus \langle D_k \rangle_1) \\ &= \langle d_k \rangle_0 \oplus \langle d_k \rangle_1 \oplus \langle D_k \rangle_0 \oplus \langle D_k \rangle_1 = d_k \oplus D_k \\ &= \mathbf{0} \oplus D_k \\ &= D_k \end{aligned}$$

Thus, the plaintext forms of all database entries is the same as it was prior to the write.

From these two cases, it is clear that the described DPF-based PIR-writing procedure will write the desired value  $v$  to the  $k$ th location in the database without changing the values written at any other location in the database.

If a prior request has written a value  $u$  to the index  $i$  where the value  $v$  is being written, the entry at that index will become corrupted and contain the value  $u \oplus v$ . In order to prevent this, most PIR writing systems first perform a query to see if any data has already been written to that entry. If an existing value  $u$  has already been written, the writer can either write to a different location or perform a write with a value of  $v \oplus u$  resulting in the value  $v = (v \oplus u) \oplus u$  being written at index  $i$ .

## 2.9 LowMC block cipher

LowMC [1] is a block cipher specially designed for secure computation settings, such as MPC protocols based on linear secret sharing, fully homomorphic encryption, and ZKPoKs. In such applications, linear operations are generally regarded as “free”, while non-linear operations (e.g., the multiplication of two or more unknowns) are costly; thus, LowMC strives to balance multiplicative complexity and depth on one hand with concrete security on the other. In the context of  $(2 + 1)$ -party computation (and associated MPC-in-the-head), multiplicative complexity dictates communication cost (transcript size) while multiplicative depth dictates round complexity (dependency-chain length for the verifier). Several tuning knobs allow protocol designers to tailor LowMC’s performance for an intended application. This flexibility and focus on efficient MPC implementation makes LowMC extremely valuable for analyzing and modifying DPF keys in MPC. In particular, implementations of the Boyle–Gilboa–Ishai DPF construction which use LowMC to construct their length-doubling PRGs can be more efficiently processed in an MPC setting. This is especially important for MPC or MPC-in-the-head verification, evaluation, or generation of DPF keys.

## Chapter 3

# Point function trees and the Boyle–Gilboa–Ishai DPF construction

In this chapter we introduce the concept of a *point function tree*. We then explain how this concept forms the basis of the Boyle–Gilboa–Ishai (BGI) DPF construction, which is the most efficient DPF construction currently available.

### 3.1 Point function trees

Consider the  $i$ th binary point function,  $P_i$ , on  $\mathbb{Z}_N$ . If we suppose, that  $N = 2^{n+k}$  for some non-negative integers  $n$  and  $k$ , then the  $i$ th point function on  $\mathbb{Z}_N$  has a natural representation as a full binary tree of height  $n$  whose  $2^n$  leaf nodes partition  $\vec{e}_i$  into  $\lambda$ -bit segments, where  $\lambda = 2^k$ . [Figure 3.1](#) illustrates this representation for the 45th point function on  $\mathbb{Z}_{2^{3+3}}$ .

This tree structure can be extended to work with generalized point functions. Consider the generalized point function  $P_{x,y}$ , where the outputs are represented as  $\ell$ -bit values. Given that  $N = 2^{n+k}$ , the tree height continues to be  $n$ . However, each of the  $2^n$  leaves now has a bitlength of  $\lambda = \ell \cdot 2^k$ . These leaves partition the vector  $y \cdot \vec{e}_i$  into  $2^n$  length  $2^k$  vectors where each entry is  $\ell$  bits in size.



In both the binary and generalized forms of the point function tree, the parameter  $k$  can be set to 0 in order to make each leaf node in the tree correspond to one output of the point function.

The above-described tree structure is the Boyle–Gilboa–Ishai DPF construction which we discuss in [Section 3.2](#). Before discussing the details of this construction, we will outline the critical concepts and terminology for point function trees.

### 3.1.1 0/1-leaves, 0/1-nodes, and 1-paths



Let  $B(P_{x,y})$  denote the height- $n$  binary-tree representation of a point function on  $\mathbb{Z}_N$  with a distinguished point  $(x, y)$ . We assign a discrete “type” to each node of  $B(P_{x,y})$  based on its pedigree. This taxonomy divides nodes into three types and also accounts for the cases where  $k > 0$ , so that each leaf node captures the image of  $2^k > 1$  consecutive inputs to  $P_{x,y}$ . This taxonomy applies to the tree representation of both binary and general point functions.

**Definition 5.** A leaf node is called a *1-leaf* if it holds a  $\lambda$ -bit scaled selection vector; it is called a *0-leaf* if it holds a  $\lambda$ -bit zero vector.

Notice that, for every point function  $P_{x,y}$ , precisely one leaf node of  $B(P_{x,y})$  is a 1-leaf and all others are 0-leaves. We colour the sole 1-leaf in [Figure 3.1](#) (and, likewise, in [Figure 3.2](#)) red (i.e., ) while all 0-leaves are green (i.e., )

The notions of 0-leaves and 1-leaves have natural, recursively defined analogues for interior nodes.

**Definition 6.** An interior node is a *0-node* if its children are both 0-leaves or both 0-nodes; it is a *1-node* if its children are either (i) a 1-leaf and a 0-leaf or (ii) a 1-node and a 0-node. The node is defined as a *2-node* if either both of its children are 1-nodes or 1-leaves, or if at least one of the children is a 2-node.

In [Figure 3.1](#), we use light shading (i.e., ) to indicate an interior node is a 0-node and dark shading (i.e., ) to indicate that it is a 1-node. Notice that, by construction, there are no 2-nodes in a well-formed point function tree. At the same time, every leaf node descendant from any 0-node is a 0-leaf whereas exactly one leaf node descendant from any 1-node is a 1-leaf (and all others are 0-leaves). From here, we can further define the notion of a *1-path* as follows.

**Definition 7.** A sequence of edges in  $B(P_{x,y})$  is a *1-path* if it originates at a 1-node and terminates at the sole 1-leaf descendent from that 1-node.

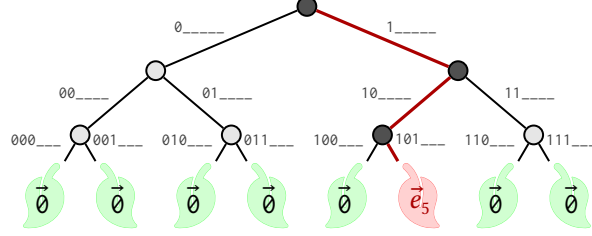


Figure 3.1: Binary-tree representation for the 45th point function on  $\mathbb{Z}_{2^{3+3}}$ . Each leaf holds either a zero vector or a selection vector of length  $2^3$ .

Notice that *every* node along a 1-path is either a 1-node or a 1-leaf. An immediate consequence of Definitions 5–7 follows in [Theorem 1](#).

**Theorem 1.** *The following three characterizations are all equivalent: A full binary tree of height  $n$  represents a point function if and only if*

1. *exactly one leaf is a 1-leaf and all others are 0-leaves;*
2. *its root is a 1-node; or*
3. *it contains a 1-path of height  $n$ .*

Consequent to Bullet 3 of [Theorem 1](#), the tree  $B(P_{x,y})$  has exactly one 1-node at each non-leaf level (and exactly one 1-leaf at the leaf level).

## 3.2 Boyle–Gilboa–Ishai DPF construction

The BGI DPF construction is the most efficient DPF design currently available. In order to reduce evaluation time and key size to be logarithmic in the domain size, BGI DPFs leverage the structure of point function trees. This is combined with the following elementary observation about pseudorandom generators (PRGs) seeded with (pseudo)randomly-sampled XOR sharings.

### 3. Point function trees and the Boyle–Gilboa–Ishai DPF construction



Figure 3.2: The BGI tree shares induced by  $[\vec{e}_{27}]$  over  $\mathbb{Z}_2^{3+3}$ .

**Observation 1** (PRGs preserve “zeroness”). *Let  $\{G_\lambda\}_{\lambda \in \mathbb{N}}$  be a length-doubling PRG family and consider  $(L, R) \leftarrow G_\lambda([z]_0) \oplus G_\lambda([z]_1)$ . If  $z = 0^\lambda$ , then  $L = R = 0^\lambda$ ; otherwise, if  $z \neq 0^\lambda$ , then both  $L \neq 0^\lambda$  and  $R \neq 0^\lambda$  with a probability overwhelming in  $\lambda$ .*

In other words, either both halves of the output are equal (because the inputs were equal), or neither half is equal (because the inputs were unequal)—at least with a very high probability. The idea from here is to use  $G_\lambda$  as a black box to build what we call a *pseudorandom traversal function*, wherein it is easy to *force* equality for a chosen half of the output while ensuring that “inadvertent equalities” in the other half remain cryptographically rare.

**Definition 8.** Let  $\{G_\lambda\}_{\lambda \in \mathbb{N}}$  be a length-doubling PRG family. The *pseudorandom traversal function family* from  $\{G_\lambda\}_{\lambda \in \mathbb{N}}$  is the infinite family  $\{\tilde{G}_\lambda\}_{\lambda \in \mathbb{N}}$  of functions  $\tilde{G}_\lambda: \mathbb{Z}_2^{\lambda-1} \times \mathbb{Z}_2 \times (\mathbb{Z}_2^{\lambda-1} \times (\mathbb{Z}_2)^2) \rightarrow \mathbb{Z}_2^\lambda \times \mathbb{Z}_2^\lambda$  such that

$$\tilde{G}_\lambda(\underbrace{s}_{\lambda-1 \text{ bits}}, \underbrace{\text{advice}}_{1 \text{ bit}}, \underbrace{cw}_{(\lambda-1)+2 \text{ bits}}) := \begin{cases} G_\lambda(s||0) & \text{if } \text{advice} = 0, \text{ and} \\ G_\lambda(s||0) \oplus (cw_L, cw_R) & \text{if } \text{advice} = 1, \end{cases}$$

where  $cw_L := \overline{cw}||t_L$  and  $cw_R := \overline{cw}||t_R$  for  $cw = (\overline{cw}, t_L, t_R)$ .

An ordered couple  $((s_0, \text{advice}_0, cw), (s_1, \text{advice}_1, cw))$  of  $\tilde{G}_\lambda$  inputs that share a common  $cw$  term is called an *input pair*. Intuitively, the  $\text{advice}_0$  and  $\text{advice}_1$  bits of an input pair indicate whether or not to “correct” the output of  $G_\lambda$  (via perturbing it by  $cw$ ) before returning it from  $\tilde{G}_\lambda$  when the PRG seed is  $s_0$  or  $s_1$ , respectively.



*Taxonomy of input pairs:* Every input pair has one of four *types*; namely, setting  $(L_0, R_0) \leftarrow \widetilde{G}_\lambda(s_0, \text{advice}_0, cw)$  and  $(L_1, R_1) \leftarrow \widetilde{G}_\lambda(s_1, \text{advice}_1, cw)$ , the pair is

1. a *0-pair* if both  $L_0 = L_1$  and  $R_0 = R_1$ ;
2. an *L-pair* if  $L_0 \neq L_1$  while  $R_0 = R_1$ ;
3. an *R-pair* if  $L_0 = L_1$  while  $R_0 \neq R_1$ ; or
4. a *2-pair* if neither  $L_0 = L_1$  nor  $R_0 = R_1$ .

When the *L*- versus *R*- “handedness” of a pair is irrelevant to the discussion (or is a secret), we refer to *L*-pairs and *R*-pairs alike as *1-pairs*. A BGI DPF tree is made up of 0-pairs and 1-pairs. 2-pairs are explicitly prevented in the DPF’s construction in order to ensure the DPF is well-formed.

At a high level, the BGI construction uses pseudorandom traversal function families to construct concise, XOR-shared binary-tree representations of point functions. To see how this works, we first examine how to construct 0- and 1-pairs for a given  $\widetilde{G}_\lambda$ .

To construct both 0-pairs and 1-pairs, we define the following terminology. We define the PRG outputs on the given seeds to be  $(\bar{L}_0, \bar{R}_0) \leftarrow G_\lambda(s_0 || 0)$  and  $(\bar{L}_1, \bar{R}_1) \leftarrow G_\lambda(s_1 || 0)$ . These can be parsed as,  $\bar{L}_b = (\tilde{L}_b, \bar{t}_{L_b}) \in \mathbb{Z}_2^{\lambda-1} \times \mathbb{Z}_2$  and  $\bar{R}_b = (\tilde{R}_b, \bar{t}_{R_b}) \in \mathbb{Z}_2^{\lambda-1} \times \mathbb{Z}_2$  for  $b \in \{0, 1\}$ . We also define  $\bar{R} := \bar{R}_0 \oplus \bar{R}_1$ , and  $\bar{L} := \bar{L}_0 \oplus \bar{L}_1$ , which are parsed as  $\bar{L} = (\tilde{L}, \bar{t}_L) \in \mathbb{Z}_2^{\lambda-1} \times \mathbb{Z}_2$  and  $\bar{R} = (\tilde{R}, \bar{t}_R) \in \mathbb{Z}_2^{\lambda-1} \times \mathbb{Z}_2$ . This also implies that  $\bar{t}_L = \bar{t}_{L_0} \oplus \bar{t}_{L_1}$  and  $\bar{t}_R = \bar{t}_{R_0} \oplus \bar{t}_{R_1}$ .

### 3.2.1 Constructing 0-pairs

Constructing a 0-pair is trivial: Choose  $(s_0, \text{advice}_0, cw)$  arbitrarily, and then set  $s_1 := s_0$  and  $\text{advice}_1 := \text{advice}_0$ . A straightforward calculation confirms that the resulting input pair is indeed a 0-pair.

Suppose that  $cw$  has been chosen arbitrarily, and that  $s_0, s_1, \text{advice}_0$ , and  $\text{advice}_1$  have all been chosen as specified. Now, we know that  $(L_0, R_0) \leftarrow \widetilde{G}_\lambda(s_0, \text{advice}_0, cw)$  and  $(L_1, R_1) \leftarrow \widetilde{G}_\lambda(s_1, \text{advice}_1, cw)$ . We are given that  $s_0 = s_1$  and  $\text{advice}_0 = \text{advice}_1$ . Thus,

$$\widetilde{G}_\lambda(s_0, \text{advice}_0, cw) = \widetilde{G}_\lambda(s_1, \text{advice}_1, cw)$$

Since  $\widetilde{G}_\lambda$  is deterministic. Therefore  $L_0 = L_1$  and  $R_0 = R_1$ , which is the definition of a 0-pair. As a result,  $\bar{t}_{L_0} = \bar{t}_{L_1}$  which implies that  $\bar{t}_L = \bar{t}_{L_0} \oplus \bar{t}_{L_1} = \bar{t}_{L_0} \oplus \bar{t}_{L_0} = 0$ , and by the same reasoning  $\bar{t}_R = 0$ .

### 3.2.2 Constructing 1-pairs

To construct a 1-pair, choose  $s_0, s_1$ , and  $advice_0$  arbitrarily subject to  $s_0 \neq s_1$ , compute  $(\bar{L}_0, \bar{R}_0) \leftarrow G_\lambda(s_0 || 0)$  and  $(\bar{L}_1, \bar{R}_1) \leftarrow G_\lambda(s_1 || 0)$ , set  $advice_1 := 1 \oplus advice_0$ ,  $\bar{R} := \bar{R}_0 \oplus \bar{R}_1$ , and  $\bar{L} := \bar{L}_0 \oplus \bar{L}_1$ , and then parse  $\bar{L}$  and  $\bar{R}$  as specified previously.

To make an  $L$ -pair, set

$$cw := (\tilde{R}, 1 \oplus \bar{t}_L, \bar{t}_R); \quad (3.1)$$

to instead make an  $R$ -pair, set

$$cw := (\tilde{L}, \bar{t}_L, 1 \oplus \bar{t}_R). \quad (3.2)$$

A slightly more involved, albeit fully mechanical, calculation confirms that in both cases the resulting input pair is indeed a 1-pair of the desired handedness.

In this calculation, we will consider the case of a  $L$ -pair. The calculations for a  $R$ -pair is completely symmetric. Now, since  $advice_0 = advice_1 \oplus 1$ , we can assume without loss of generality that  $advice_0 = 0$  and  $advice_1 = 1$ . We then have that  $L_0 = \bar{L}_0 \oplus cw_L$  and  $L_1 = \bar{L}_1$ . Similarly,  $R_0 = \bar{R}_0 \oplus cw_R$  and  $R_1 = \bar{R}_1$ . This gives the following for the right child

$$\begin{aligned} R_0 &= \bar{R}_0 \oplus cw_R \\ &= \bar{R}_0 \oplus (\overline{cw} || t_R) \\ &= \bar{R}_0 \oplus (\tilde{R} || t_R) \\ &= (\tilde{R}_0 || \bar{t}_{R_0}) \oplus (\tilde{R} || t_R) \\ &= (\tilde{R}_0 \oplus \tilde{R}) || (\bar{t}_{R_0} \oplus t_R) \\ &= (\tilde{R}_0 \oplus (\tilde{R}_0 \oplus \tilde{R}_1)) || (\bar{t}_{R_0} \oplus (\bar{t}_{R_0} \oplus \bar{t}_{R_1})) \\ &= \tilde{R}_1 || \bar{t}_{R_1} \\ &= R_1 \end{aligned}$$

This implies that  $\bar{t}_{R_0} = \bar{t}_{R_1}$  meaning that  $\bar{t}_R = \bar{t}_{R_0} \oplus \bar{t}_{R_1} = \bar{t}_{R_0} \oplus \bar{t}_{R_0} = 0$ .

For the left child, the result is

$$\begin{aligned}
 L_0 &= \bar{L}_0 \oplus cw_L \\
 &= \bar{L}_0 \oplus (\bar{cw}||t_L) \\
 &= \bar{L}_0 \oplus (\tilde{R}||t_L) \\
 &= (\tilde{L}_0||\bar{t}_{L_0}) \oplus (\tilde{R}||t_L) \\
 &= (\tilde{L}_0 \oplus \tilde{R})||(\bar{t}_{L_0} \oplus t_L) \\
 &= (\tilde{L}_0 \oplus \tilde{R})||(\bar{t}_{L_0} \oplus (1 \oplus t_L)) \\
 &= (\tilde{L}_0 \oplus (\tilde{R}_0 \oplus \tilde{R}_1))||(\bar{t}_{L_0} \oplus (1 \oplus \bar{t}_{L_0} \oplus \bar{t}_{L_1})) \\
 &= (\tilde{L}_0 \oplus (\tilde{R}_0 \oplus \tilde{R}_1))||(\bar{t}_{L_0} \oplus (1 \oplus \bar{t}_{L_1}))
 \end{aligned}$$

Now,  $L_1 = \bar{L}_1 = (\tilde{L}_1, \bar{t}_{L_1})$ . In order to have  $L_0 = L_1$ , we must have  $\tilde{L}_1 = \tilde{L}_0 \oplus (\tilde{R}_0 \oplus \tilde{R}_1)$ . However,  $\tilde{L}_1$  is pseudorandom by its definition, and  $\tilde{L}_0 \oplus (\tilde{R}_0 \oplus \tilde{R}_1)$  is pseudorandom, since it is the XOR of three pseudorandom values. Thus, the probability that  $\tilde{L}_1 = \tilde{L}_0 \oplus (\tilde{R}_0 \oplus \tilde{R}_1)$  is  $\frac{1}{2^{\lambda-1}}$ . Since  $2^{\lambda-1}$  is an exponential function,  $\lambda^a \in o(2^{\lambda-1})$  for all  $a \in \mathbb{R}^+$ . As a result, the probability  $\frac{1}{2^{\lambda-1}}$  is negligible in the security parameter  $\lambda$ . At the same time,  $\bar{t}_{L_1} \neq 1 \oplus \bar{t}_{L_0} = \bar{t}_{L_0}$  by definition. Thus,  $\bar{t}_L = \bar{t}_{L_0} \oplus (1 \oplus \bar{t}_{L_1}) = 1$ .

### 3.2.3 Chaining 1-pairs

Notice that 0-pairs are agnostic to the values of  $cw$  and  $advice_b$  (provided  $advice_1 = advice_0$  holds), whereas 1-pairs require a very specific choice for  $cw$  (i.e., one that depends on  $s_0, s_1$ , and the desired handedness) and also that  $advice_0 = 1 \oplus advice_1$ . We recast part of this as a formal observation, to be used in the next chapter.

**Observation 2.** *If  $((s_0, advice_0, cw), (s_1, advice_1, cw))$  is a  $b$ -pair for  $b \in \{0, 1\}$ , then  $advice_0 \oplus advice_1 = b$ .*

Equations (3.1) and (3.2) together ensure the nonzero half of  $\tilde{G}_\lambda(s_0, advice_0, cw) \oplus \tilde{G}_\lambda(s_1, advice_1, cw)$  has a 1 as its rightmost bit so that parsing that half of  $\tilde{G}_\lambda(s_0, advice_0, cw)$  and  $\tilde{G}_\lambda(s_1, advice_1, cw)$  each as  $\mathbb{Z}_2^{\lambda-1} \times \mathbb{Z}_2$  elements yields a pair of values suitable for constructing another 1-pair. This enables the natural construction of 1-pair chains consisting of the initial  $(s_b, advice_b)$  values linked by an array of  $cw$  terms.

### Chaining 1-pairs example

Consider the bitstring  $x = 101$ . We want to build a 1-pair chain that corresponds to this bitstring.

- Select the initial seeds,  $s_0^0$  and  $s_1^0$ , and advice bit values,  $advice_0$  and  $advice_1$ , as specified in [Section 3.2.2](#).
- We then construct  $cw^0$  so that the first pair is a  $R$ -pair.
- Now, the resulting right children are  $R_0^1 \neq R_1^1$ .
- Parse the right children as  $R_0^1 = (s_0^1, advice_0^1)$  and  $R_1^1 = (s_1^1, advice_1^1)$ .
- As shown in [Section 3.2.2](#),  $s_0^1 \neq s_1^1$  and  $advice_0^1 = 1 \oplus advice_1^1$ .
- Since the seeds and advice bits meet the requirements for constructing a 1-pair, construct  $cw^1$  so that the resulting pair is a  $L$ -pair.
- Now, the resulting left children are  $L_0^2 \neq L_1^2$ .
- Parse the left children as  $L_0^2 = (s_0^2, advice_0^2)$  and  $L_1^2 = (s_1^2, advice_1^2)$ .
- As shown in [Section 3.2.2](#),  $s_0^2 \neq s_1^2$  and  $advice_0^2 = 1 \oplus advice_1^2$ .
- Since the seeds and advice bits meet the requirements for constructing a 1-pair, construct  $cw^2$  so that the resulting pair is a  $R$ -pair.
- Now, the resulting right children are  $R_0^3 \neq R_1^3$ , which can be used as seeds for later rounds of 1-pair chaining if needed.

This demonstrates how a chain of 1-pairs can be constructed in a chain based upon a specified point  $x$ .

### BGI share generation

At a high level, the BGI construction assembles a chain of 1-pairs whose handedness reflect the leftmost bits of the distinguished input. We note the similarities between the construction of this chain and the definition of the 1-path in the corresponding point-function tree (cf. [Theorem 1](#)). Suppose the distinguished point is  $(x, y)$  where

$x \in [0 \dots N)$  and  $y \in [0 \dots 2^\ell)$ , given  $N = 2^{n+k}$  and  $\lambda = \ell \cdot 2^k$ . The chain begins with a uniformly random  $L$ -pair if the leftmost bit of  $x$  is 0 and a uniformly random  $R$ -pair if it is 1; the next link is an  $L$ -pair if the second-leftmost bit of  $x$  is 0 and an  $R$ -pair if it is 1, and so on until the chain accounts for each of the leftmost  $n$  bits of  $i$ .

For the remaining  $k$  bits, let  $x' = x \bmod 2^k$  and let  $y \cdot \vec{e}_{x'}$  be the scaled selection vector of length  $2^k$  in which the  $x'$ th entry is the  $\ell$ -bit value  $y$  and all other entries are  $\ell$ -bit zero values. Suppose  $(\bar{L}_0, \bar{R}_0)$  and  $(\bar{L}_1, \bar{R}_1)$  are output by the last 1-pair in the chain. If that last pair is an  $L$ -pair, then output  $leaf := \bar{L}_0 \oplus \bar{L}_1 \oplus y \cdot \vec{e}_{x'}$ ; otherwise, if it is an  $R$ -pair, then output  $leaf := \bar{R}_0 \oplus \bar{R}_1 \oplus y \cdot \vec{e}_{x'}$ .

Since the outputs  $(\bar{L}_0, \bar{R}_0)$  and  $(\bar{L}_1, \bar{R}_1)$  are produced as the output of a PRG with security parameter  $\lambda$ , the size of a leaf node is  $\lambda$ . In case the final output needs to be larger, a final PRG can be used before the leaf correction is applied. In this case, the leaf layer correction is  $leaf := S(\bar{V}_0) \oplus S(\bar{V}_1) \oplus y \cdot \vec{e}_{x'}$ , where  $V_0$  and  $V_1$  are the shares of the leaf 1-pair in the DPF tree and  $S: \mathbb{Z}_\lambda \rightarrow \mathbb{Z}_{2^\ell}$  is a PRG which produces outputs of the necessary length.

Each DPF share  $\llbracket (x, y) \rrbracket_b$  then consists of (i) the  $b$ th share of the 1-pair chain (i.e.,  $(s_b, advice_b)$  and the array of  $n$  correction terms) alongside (ii) the final *leaf* value.

### From 1-pair chains to BGI tree shares

Owing to the pseudorandomness of  $G_\lambda$ , it is computationally infeasible for a shareholder knowing only one of  $\llbracket (x, y) \rrbracket_0$  or  $\llbracket (x, y) \rrbracket_1$  to deduce the sequence of  $L$ - versus  $R$ -ward traversals (i.e., the leftmost bits of  $x$ ) reflected in the 1-pair chain. Nevertheless, such a shareholder can “evaluate” the chain for *any* of the  $2^n$  distinct length- $n$  traversal sequences. Consequently, we can think of the 1-pair chain as implicitly defining a (componentwise-)XOR-shared *full binary tree* of height  $n$ , which we refer to as the *BGI tree induced by  $\llbracket (x, y) \rrbracket$* . For a binary DPF this is referred to as the BGI tree induced by  $\llbracket \vec{e}_x \rrbracket$ .

If ever the traversal sequence diverges from the binary representation of the distinguished input  $x$ , then, by definition, it transits (one share of) a 0-pair. Consequently, if both shareholders evaluate their respective shares on that same sequence, then *the pseudorandom values they produce must coincide from that 0-pair onward*—up

to and including the leaf node. In particular, we have just argued that (i) corresponding leaves *not* at the end of the 1-pair chain hold XOR-shares of  $0^\lambda$ ; furthermore, due to the way the value *leaf* was constructed, (ii) corresponding leaves that *are* at the end of the 1-pair chain hold XOR-shares of  $y \cdot \vec{e}_{x'}$ , with  $x' = x \bmod 2^k$  capturing the  $k$  rightmost bits of the distinguished input  $x$ .

**Observation 3.** *Let  $(\langle D \rangle_0, \langle D \rangle_1)$  be the BGI tree shares induced by  $\llbracket (x, y) \rrbracket$ . If corresponding nodes in  $\langle D \rangle_0$  and  $\langle D \rangle_1$  form a 0-pair, then their reconstructed counterpart in  $D$  is a 0-node (or a 0-leaf); likewise, if they form a 1-pair, then their reconstructed counterpart in  $D$  is a 1-node (or a 1-leaf).*

The shareholders can, therefore, “evaluate” their respective DPF shares at any input  $j \in [0 \dots 2^{n+k})$  to obtain an XOR-sharing  $(P_{x,y}(j))$  by applying  $\tilde{G}_\lambda$  repeatedly (with appropriate handedness) until arriving at the leaf, and then extracting the desired  $\ell$  bits from that leaf.

Figure 3.2 illustrates the BGI tree shares induced by  $\llbracket \vec{e}_{27} \rrbracket$  over  $\mathbb{Z}_{2^{3+3}}$ . The single bit set off in a box to the right of each node is that node’s *advice* bit; edges are doublestruck (i.e., ‘ $\mathbb{I}$ ’) if the advice bit of the parent is 1 (a perturbation by *cw* is applied) and singlestruck (i.e., ‘ $\mathbb{I}$ ’) otherwise (no perturbation by *cw* is applied). The 1-pair chain is set off in **red**. In between the two tree shares, we draw the reconstructed leaf nodes including the 1-byte substring of  $\vec{e}_x$  that each leaf holds.

We conclude this section by asserting that the BGI construction constitutes a  $(2, 2)$ -DPF with 1-bit outputs (cf. Definition 4).

**Theorem 2** ([13; Theorem 3.3]). *If  $\{G_\lambda\}_{\lambda \in \mathbb{N}}$  is a length-doubling PRG family, then the BGI construction is a  $(2, 2)$ -DPFs with  $\ell$ -bit outputs. Each DPF share comprises just  $n(\lambda + 2) + \log_2 |\mathbb{G}| \in O(\lg N)$ , where  $\mathbb{G}$  is the output group. Assuming outputs are  $\ell$ -bit values,  $|\mathbb{G}| = 2^\ell$ , so keys are of size  $(n + k) \cdot (\lambda + 2) + \ell \in O(\lg N) \in O(\lg N)$ .*

*In the special case outputs are one bit in length (i.e.  $\ell = 1$ ), the key size is reduced to approximately  $n \cdot \lambda$  by packing  $\lambda = 2^k$  DPF outputs into each leaf of the DPF tree [13].*

Interested readers can find proof of Theorem 2 and additional details about the BGI construction in Boyle et al.’s manuscript [13; §3.2.2].

## Chapter 4

# Parity-segment trees and DPFs

In this chapter, we introduce the parity-segment tree data structure, and we explore its relation to the Boyle–Gilboa–Ishai DPF construction.

### 4.1 Parity-segment trees

A *parity-segment tree* is a data structure for answering parity queries over the substrings of a binary string, with a worst-case complexity logarithmic in the bitstring’s length. That is, given the parity-segment tree  $T(x)$  for a length- $N$  bitstring  $x = x_0x_1 \cdots x_{N-1}$ , a parity query for the substring  $x[a..b)$  of  $x$ ,  $0 \leq a < b \leq N$ , returns the parity

$$\text{parity}(x[a..b)) := \bigoplus_{i=a}^{b-1} x_i$$

with a running time in  $O(\lg N)$ .

Constructing the parity-segment tree  $T(x)$  for a given bitstring  $x$  is straightforward, if tedious. For ease of exposition, suppose that  $x$  has length  $N = 2^{n+k}$  for some nonnegative integers  $n$  and  $k$  and that each leaf node represents  $\lambda = 2^k$  consecutive bits of  $x$ . Notice that  $\lambda \mid N$  by construction.

Given  $x$ , we construct the tree  $T(x)$  from the bottom up: To form the base of the tree, split  $x$  into  $N/\lambda = 2^n$  many  $\lambda$ -bit substrings and then insert one leaf node for each of these substrings, storing its parity inside the node. Next, for each successive pair of leaf nodes, insert a parent and store within it the combined parity of its two

children. Now repeat this process on each successive pair of parents, and so on, until a single root emerges (after  $n - 1$  recursions). Notice that the parity bit held by any given node equals the parity of a concatenation over all substrings of  $x$  associated with leaves descendant from that node.

It follows easily by inspection that constructing the parity-segment tree  $T(x)$  from  $x$  requires the computation of  $(N/\lambda)$ -many  $\lambda$ -bit parities (for the leaf nodes) plus  $2^n - 1$  single-bit XORs (for the interior nodes), giving a total complexity of  $O(N)$  bit operations; the tree itself occupies  $2^{n+1} - 1 \in O(N/\lambda)$  bits.

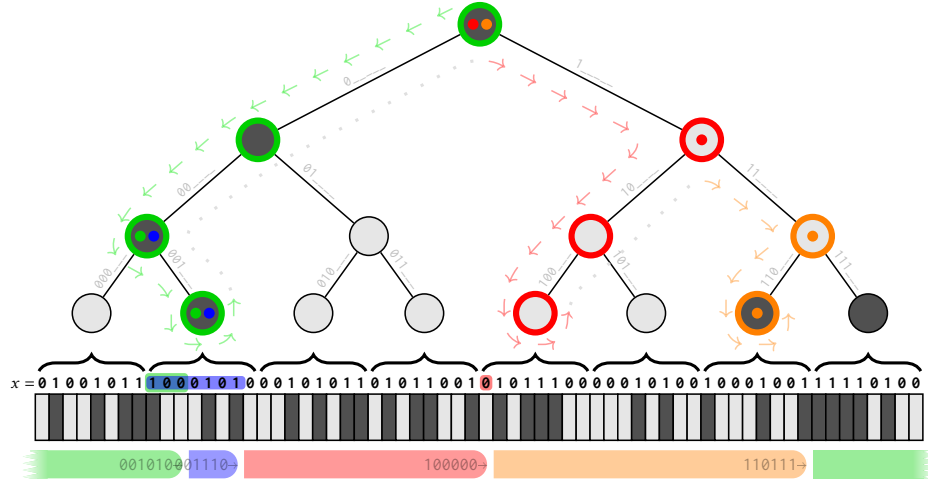


Figure 4.1: The parity-segment tree for a 64-bit string with 8-bit leaf nodes (i.e.,  $N = 2^{3+3}$  and  $\lambda = 2^3$ ).

#### 4.1.1 Computing segment parities

Figure 4.1 shows the parity-segment tree for an arbitrary 64-bit string  $x$ , which is written immediately below the tree. In this toy example, each leaf node is associated with a 1-byte (8-bit) substring—yielding  $64/8 = 8$  leaf nodes in total—and holds the parity of that substring internally. Likewise, each non-leaf node holds 1 bit indicating the parity of its immediate descendants. We use light shading (e.g.,  $\bigcirc$ ) to indicate a node holds even parity (the bit is 0) and dark shading (e.g.,  $\bullet$ ) to indicate it holds odd parity (the bit is 1).





Beneath the bitstring, we draw several half-open segments that collectively partition the bitstring into four contiguous (up to cyclic rotation) substrings, totally



ordered by their rightmost endpoints. With 0-based indexing, the first (**green**) segment ends after bit 10; the second (**blue**) segment after bit 14; the third (**red**) segment after bit 32; and the fourth (**orange**) segment after bit 55.

The diagram also contains several pictorial annotations conveying information about how the *prefix-parity algorithm* helps to find the parity of each segment. As its name suggests, the prefix-parity algorithm computes the parity of each substring using the parities of *prefixes* of  $x$  sharing the same rightmost endpoints as the desired segments. In this example, it finds each of the prefix parities  $\text{parity}(x[0..11])$ ,  $\text{parity}(x[0..15])$ ,  $\text{parity}(x[0..33])$ , and  $\text{parity}(x[0..56])$ . A subsequent post-processing phase exploits the nilpotency of XOR to compute the desired segment parities from these prefix parities via

$$\begin{aligned}
 \text{parity}(\text{green}) &= \text{parity}(x[56..64]) \oplus \text{parity}(x[0..11]) \\
 &= (\text{parity}(x) \oplus \text{parity}(x[0..56])) \\
 &\quad \oplus \text{parity}(x[0..11]); \\
 \text{parity}(\text{blue}) &= \text{parity}(x[11..15]) \\
 &= \text{parity}(x[0..15]) \oplus \text{parity}(x[0..11]); \\
 \text{parity}(\text{red}) &= \text{parity}(x[15..33]) \\
 &= \text{parity}(x[0..33]) \oplus \text{parity}(x[0..15]); \text{ and} \\
 \text{parity}(\text{orange}) &= \text{parity}(x[33..56]) \\
 &= \text{parity}(x[0..56]) \oplus \text{parity}(x[0..33]).
 \end{aligned}$$

In the diagram, a node is drawn with a thick coloured outline (e.g., ) if the prefix-parity algorithm visits that node during the computation of one or more prefix parities. We employ a memoization (and backtracking) strategy that ensures each node is visited at most once throughout the computation of all prefix parities; the outline's colour indicates which prefix the algorithm is computing when it *first* visits that node. The dashed-and-dotted path emanating from the root likewise shows the traversal order through the tree; we decorate the path with coloured-arrow dashes (e.g., ) when the traversal is visiting new nodes and with faint gray dots (e.g., ) when “backtracking” to a previously visited (memoized) node. We place a thick coloured dot (e.g., ) within a node if the 1-bit parity stored at that node appears as an operand when computing the prefix parity for the correspondingly

coloured segment; moreover, we highlight a prefix of the  $\lambda$ -bit substring associated with a leaf if the parity of that prefix also appears as an operand in the prefix-parity computation.

The prefix-parity algorithm performs a binary search-like traversal through  $T(x)$ , employing a simple inclusion-exclusion strategy to compute a sequence of parities of prefixes that alternately over- and undershoot the desired prefix. By adopting the convention that one always “traverses left” both (i) to arrive at the root and (ii) to access the substring associated with a leaf node, we obtain the following procedure:

1. initialize a “running parity” to 0;
2. starting from the root, traverse to the leaf node associated with the rightmost bit in the prefix;
3. wherever the root-to-leaf path changes directions, update the running parity by XORing in the parity stored at the node where the change-of-direction occurs; and
4. finally, XOR in any bits of the prefix that reside in the substring associated with the leaf node.

As a modest optimization, one can terminate the traversal early if ever the rightmost endpoint is one bit past the end of the rightmost descendant of the left child of the node presently being traversed (as in such cases, the running parity is already guaranteed to be correct).

Annotated C-like pseudocode for the prefix-parity algorithm—incorporating both the above *early-termination optimization* as well as the bookkeeping needed for effective memoization—is shown in [Figure 4.2](#).

### Illustrated walkthroughs for [Figure 4.1](#)

We strategically chose the segments in [Figure 4.1](#) to illustrate some notable sub-cases, namely (i) cyclically wrapping segments (**green**), (ii) two segments terminating at the same leaf node (**green** and **blue**), (iii) segments terminating immediately following a leaf node (**orange**), and (iv) the “typical” case where a segment is alone

## 4. Parity-segment trees and DPFs

```

1 // Computes the prefix parities for each of the given endpoints
2 //
3 // Parameters:
4 //   bound - a sorted list of segment endpoints
5 //   T      - a parity-segment tree
6 //   n,k    - depth of tree, lg(bits per leaf)
7 //
8 vector prefix_parities(vector bound, tree T, int n, int k)
9 {
10     vector res           = {} // to be populated with prefix parities
11     vector path          = {T.root} // memoized current path from the root
12     vector direction     = {} // traversal directions along path
13     // 0 is to the left; 1 is to the right
14     vector parity        = {} // cumulative prefix parity along path
15     vector left          = {2*(n-1)} // leftmost-beneath-right-child along path
16     int prev            = ~bound[0] // *complement* of first shifted bound;
17     // ensures from=0 in first loop iteration
18
19     for (int i = 0; i < bound.length; i++)
20     {
21         int from = clz(bound[i] ^ prev) // common prefix length
22         int to   = n // by default, traverse entire depth
23
24         for (int j = from; j < to; j++) // iterate only over (non-common) suffix
25         {
26             int next_dir = (left[j] <= bound[i]) ? 1 : 0 // going right or left?
27             int next_path = T.traverse(path[j], next_dir) // go that direction
28             int next_parity = (next_dir == direction[j]) // update running parity
29                 ? parity[j] // no change
30                 : path[j].parity ^ parity[j] // include/exclude
31             int next_left = (next_dir == 1) // update leftmost bit
32                 ? left[j] + 2*(64-2-j) // advance right
33                 : left[j] - 2*(64-2-j) // unadvance left
34             //^^^^^^^^^^^^ <- #bits beneath each child
35
36             path[j+1] = next_path // memoize node along path
37             direction[j+1] = next_dir // memoize direction of traversal
38             left[j+1] = next_left // memoize leftmost-beneath-right-child
39             parity[j+1] = next_parity // memoize cumulative parities
40
41             if (next_left == bound[i]) // early-termination optimization
42                 to = j // -> halt traversal at current level
43         } // inner for loop ends
44
45         res[i] = parity[to] // record running prefix-parity
46         if (to == n) // conditionally add partial-leaf parity
47         {
48             string substr = path[n].substr // substring associated with leaf
49             int prefix_len = bound[i] % 2*k // length of prefix to compute
50             res[i] ^= prefix_parity_str(substr, prefix_len)
51         }
52         prev = bound[i] // for computing common prefix length
53     } // outer for loop ends
54     return res // n.b.: *prefix* (not segment) parities!
55 } // prefix_parities

```

**vector** path memoizes each node along the path from the root to the leaf node whose substring contains the bound[i]th bit, sidestepping the need to revisit any node in the common prefix between bound[i] and bound[i+1] (and, likewise, between bound[i-1] and bound[i]). If **vector** bound is lexicographically sorted, then this is sufficient to ensure that no edge in the tree is traversed more than once.

Similarly, **vector** parity memoizes running parities along this path. Since the "inclusion-exclusion" decisions used to update running parities depend on traversal direction *changes*, we also use **vector** direction to memoize traversal directions along this path. Meanwhile, **vector** left exists to assist in deciding which direction to go.

**int** from is the index of the first bit *after* the common prefix; i.e., the point starting from which memoized values reflect prev but not bound[i]. In the first loop iteration, prev == ~bound[i], which ensures that from=0. On the subsequent iterations, from is set to clz(bound[i] ^ prev), the number of leading zeros in the binary representation of bound[i] ^ prev (i.e., the number of prefix bits common to the current and previous bound). **int** to=n at the start of each loop iteration, but it may be reduced by the early-terminate optimization on Lines 41-42.

**int** next\_dir indicates whether to traverse to the right (1) or left (0), depending on whether parity[j] currently over- or undershoots path[j]. After traversing in that direction on Line 27, Lines 28-33 compute the next\_parity and the next\_left. The ternary operator on lines 28-30 carries forward parity[j] if there was no change in traversal direction; otherwise, it first "updates" that parity by XORing in the parity of the node just traversed. All four values are memoized on Lines 36-39.

If, by serendipity, path[j+1] neither overshoots nor undershoots bound[i] at this point, then parity[j+1] already equals the desired parity and we can break from the inner loop now—even if we haven't yet reached a leaf. We break by setting to=j so that Lines 45-51 can easily determine whether or not we manually broke from the inner loop.

If we did *not* manually break from the inner loop, then we must complete the parity computation by XORing in some prefix of the substring in the leaf node currently stored in path[n].

Figure 4.2: C-like pseudocode listing for the prefix-parity algorithm (left) with running commentary (right).

in terminating partway through some leaf (**red**). We remark on the implications of these cases in the algorithm walkthroughs below.

**Prefix 1 (green):** The first segment ends 3 bits into the second leaf node. Our algorithm traverses leftward twice to arrive at the parent of that node. Since the next traversal goes right (a change of direction), it reads the parity bit (odd) within that parent. After traversing right to the second leaf node, it must traverse left to access the associated substring; thus, it XORs in that leaf node's parity bit (also odd). Finally, it inspects the substring beneath that leaf node,

Table 4.1: The sequences of prefixes (and associated parities) of  $x$  considered (accounting for memoization) while computing the four prefix parities arising in Figure 4.1.

level	green	blue	red	orange
root $\rightarrow$ 0	$x[0..0) \circ$	$x[0..0) \circ$	$x[0..64)\bullet$	$x[0..64)\bullet$
1	$x[0..0) \circ$	$x[0..0) \circ$	$x[0..32)\bullet$	$x[0..64)\bullet$
2	$x[0..16)\bullet$	$x[0..16)\bullet$	$x[0..32)\bullet$	$x[0..48)\bullet$
leaf $\rightarrow$ 3	$x[0..8) \circ$	$x[0..8) \circ$	$x[0..32)\bullet$	$x[0..56)\bullet$
substring $\rightarrow$ 4	$x[0..11)\bullet$	$x[0..15)\bullet$	$x[0..33)\bullet$	—

XORing in the parity of its 3-bit prefix (odd yet again). The resulting parity is therefore

$$\begin{array}{rcl}
 \text{parity}(x[0..11)) & = & \text{parity}(01001011\ 10001010) & 1 \\
 & \oplus & \text{parity}(00000000\ 10001010) & \oplus 1 \\
 & \oplus & \text{parity}(00000000\ 10000000) & \oplus 1 \\
 \hline
 & = & \text{parity}(01001011\ 10000000) & = 1.
 \end{array}$$

In total, the algorithm visits four nodes (and examines one  $\lambda$ -bit substring) to compute this prefix parity.

**Prefix 2 (blue):** The second segment also ends part of the way through the second leaf; consequently, the algorithm reuses almost the entire first parity computation, merely substituting in a longer substring prefix in the last step. In total, the algorithm visits zero new nodes (and examines one  $\lambda$ -bit substring) to compute this prefix parity.

**Prefix 3 (red):** The third segment extends one bit into the right subtree of the root (a direction change at the outset). Hence, the algorithm XORs the parity stored in the root (which captures the parity of the *entire* string  $x$ ) together with the parity stored within the root's right child (which captures the parity of the *second half* of  $x$ ), after which it holds the parity of the first half of  $x$ . For the remaining bit, it traverses to the leftmost leaf beneath the right child of the root (which involves no direction changes), and examines the single bit of the associated substring that is part of the segment. In total, the algorithm visits

three new nodes (and examines one  $\lambda$ -bit substring) to compute this prefix parity.

**Prefix 4 (orange):** The fourth prefix terminates immediately after the substring in the second-last leaf node. It XORs the memoized 1-bit parity from the root together with the parity stored within the second-last leaf node and its parent, after which it holds the desired parity. In total, the algorithm visits two new nodes (and does not examine any substring) to compute this prefix parity.

All told, the prefix-parity algorithm in this example visited  $4 + 0 + 3 + 2 = 9$  (out of 15) nodes and examined  $1 + 0 + 1 + 0 = 2$  (out of 8) distinct  $\lambda$ -bit substrings of  $x$  (computing  $1 + 1 + 1 + 0 = 3$  substring-prefix parities).

Table 4.1 lists the sequence of prefixes of  $x$  whose parities are computed (and memoized) as the prefix-parity algorithm computes the above four prefix parities. In the table, segment parities are coloured gray wherever a memoized value is in use; the arrows indicate where each memoized value was most recently used.

#### 4.1.2 Analysis

Our primary concern in the sequel will be how many distinct *edges* the prefix-parity algorithm must traverse—and, to a lesser extent, how many  $\lambda$ -bit *substrings* it must examine—for a given partitioning of  $x$  into segments. The next theorem characterizes the worst-case cost for the number of edges (assuming optimal memoization).

**Theorem 3.** *Given a parity-segment tree  $T(x)$  of height  $n$  and a lexicographically sorted list  $E$  of  $S$  distinct prefix endpoints, the prefix-parity algorithm traverses at most  $Sn - \sum_{i=2}^S \lceil \lg(i-1) \rceil$  edges to compute all  $S$  prefix parities.*

Before proving Theorem 3, we state and prove the following lemma based on the intuition that a “worst-case” instance for the prefix-parity algorithm is one that minimizes the lengths of common prefixes in (the binary representations of) successive endpoints in  $E$ .

**Lemma 1.** *If  $P$  is a set of  $S$  distinct bitstrings, then*

$$\sum_{x \in P} |x| \geq \sum_{i=1}^{S-1} \lceil \lg(i+1) \rceil, \quad (4.1)$$

where  $|x|$  denotes the length (in bits) of  $x$ .

*Proof (Sketch).* We first note that equality holds in Equation (4.1) when  $P$  consists of the empty string together with the binary representations of all non-negative integers less than  $S - 1$  [58]. Moreover, substituting one or more bitstrings from  $P$  with the binary representations of integers greater than or equal to  $S - 1$  results in a set whose aggregate length is likewise greater than or equal to that of  $P$ .  $\square$

*Proof of Theorem 3.* The result follows by induction on  $S$ .

**Base case ( $S = 1$ ):** This case is immediate by inspection.

**Inductive step:** Assume the prefix-parity algorithm traverses at least  $(S - 1)n - \sum_{i=2}^{S-1} \lfloor \lg(i - 1) \rfloor$  edges for  $S - 1$  endpoints and let  $E$  be some length- $(S - 1)$  sequence of endpoints inducing this worst-case cost. We will construct a worst-case length- $S$  sequence  $E'$  by inserting one additional endpoint at an appropriate (sorted) position within  $E$ . Specifically, to ensure that the resulting sequence is also a worst-case sequence, it suffices to choose a position among existing endpoints that (globally) minimizes the length of its common prefix with its immediate neighbours in the resulting ordered sequence (thereby maximizing the number of new edges to traverse).

Thus, we construct  $E'$  by inserting an arbitrary endpoint whose prefix is one of the (not necessarily unique) shortest prefixes not yet reflected in  $E$ . Now, consider the sets  $P$  and  $P'$  of shortest unique prefixes for  $E$  and  $E'$ , respectively. From Lemma 1, we have

$$\begin{aligned} \sum_{x \in P'} |x| - \sum_{x \in P} |x| &\geq \sum_{i=1}^{S-1} \lceil \lg(i + 1) \rceil - \sum_{i=1}^{S-2} \lceil \lg(i + 1) \rceil \\ &= \lceil \lg(S - 1 + 1) \rceil \\ &\geq \lfloor \lg S \rfloor. \end{aligned}$$

As all but the last bit of the shortest unique prefix corresponds to an already-traversed edge, this newly added endpoint necessitates traversing at most

$$n - (\lfloor \lg S \rfloor - 1) \geq n - \lfloor \lg(S - 1) \rfloor$$

additional edges, for a total number of edges traversed of at most

$$\begin{aligned} & ((S-1)n - \sum_{i=2}^{S-1} \lfloor \lg(i-1) \rfloor) + (n - \lfloor \lg(S-1) \rfloor) \\ &= Sn - \sum_{i=2}^S \lfloor \lg(i-1) \rfloor. \end{aligned}$$

□

**Theorem 3** implies that the prefix-parity algorithm visits  $o(Sn)$  edges to compute  $S$  prefix parities (of a given  $2^{n+k}$ -bit string). We also note that tree  $T(x)$  has just  $2^{n+1} - 2$  edges in total, which yields another upper bound on the number of edges traversed. Notably, as  $S$  approaches the length  $N = 2^{n+k}$  of  $x$ , the amortized number of edges traversed per prefix tends to  $2^{-k+1}$  as each of the  $2^{n+1} - 2$  edges is traversed exactly once (owing to memoization).<sup>1</sup>

The theorem deals with worst-case costs. The *expected* number of edges traversed depends on the distribution of the prefixes. Generally speaking, more densely packed prefix endpoints (i.e., shorter segments) imply greater amortization savings.

### Expected savings from early termination

Notice that the “early-termination” optimization saves exactly  $i + 1$  traversals (this includes the “traversal” from a leaf node to its associated substring) if and only if the endpoint is a multiple of  $2^i \cdot \lambda$  but not of  $2^{i+1} \cdot \lambda$ . The next theorem follows easily from this observation.

**Theorem 4.** *For a uniform random endpoint  $X \in [0..N)$ , the early-termination optimization saves  $(2 - 2^{-n})/\lambda$  traversals in expectation.*

*Proof.* For  $i = 1, \dots, n-1$ , the probability that uniform  $X$  is a multiple of  $\lambda \cdot 2^i$  but not  $\lambda \cdot 2^{i+1}$  is given by  $\frac{1}{\lambda \cdot 2^i} - \frac{1}{\lambda \cdot 2^{i+1}} = \frac{1}{\lambda \cdot 2^{i+1}}$ ; for  $i = n$ , it is just  $\frac{1}{\lambda \cdot 2^n}$ . Hence, in expectation, we save

$$\frac{n+1}{\lambda \cdot 2^n} + \sum_{i=0}^{n-1} \frac{i+1}{\lambda \cdot 2^{i+1}} = \frac{2 - 2^{-n}}{\lambda}$$

traversals (counting “traversals” from leaf nodes to substrings). □

<sup>1</sup>Indeed, it is easy to check that  $Sn - \sum_{i=2}^S \lfloor \lg(i-1) \rfloor = 2^{n+1} - 1$  in this case.

## 4.2 Selection vector rotation

While selection vectors, and, by extension, DPFs, have many useful properties, existing protocols for producing a secret shared selection vector  $[\vec{e}_i]$  when the value  $i$  is itself additively secret shared require  $O(N)$  computation and communication costs. To address this, we propose a method for using precomputation to more efficiently generate selection vectors in the online phase of an MPC protocol using only one round of communication. Since DPFs are simply secret shared, “functional” representations of a selection vector, the techniques we discuss for preprocessing and modifying selection vectors can be adapted for use with DPFs.

We begin with the following observation.

**Observation 4.** *All selection vectors of a given length are equivalent up to cyclic rotation. Specifically, for any  $i, j \in [0 \dots N)$ , if  $\vec{e}_i$  is the  $i$ th selection vector of length  $N$ , then  $\vec{e}_j = \vec{e}_i \gg (j - i)$  is the  $j$ th selection vector of length  $N$ .*

Observation 4 is especially relevant when the selection vector is secret shared: If  $P_0$  and  $P_1$  hold additive sharings  $[i]$  and  $[j]$  of two numbers from the ring of integers modulo  $N$  alongside a sharing of the  $i$ th selection vector of length  $N$ , then they can arrive at a sharing of the  $j$ th selection vector of length  $N$  as follows. First, they leverage linearity to learn

$$\begin{aligned} (j - i) \bmod N &= (([j]_0 + [j]_1) - ([i]_0 + [i]_1)) \bmod N \\ &= \underbrace{([j]_0 - [i]_0)}_{P_0 \text{ shares}} + \underbrace{([j]_1 - [i]_1)}_{P_1 \text{ shares}} \bmod N \end{aligned} \quad (4.2)$$

without revealing  $i$  or  $j$  individually, after which each party cyclically rotates its own share of  $\vec{e}_i$  to the right by this quantity.<sup>2</sup> Notice that if  $i$  is uniform, then  $(j - i) \bmod N$  perfectly hides  $j$ , making this transformation from the  $i$ th into the  $j$ th selection vector perfectly oblivious.

<sup>2</sup>This approach is mathematically sound for additive sharings because cyclic rotation is a linear operation, namely multiplication by a cyclic permutation matrix (i.e., a cyclic rotation of the identity matrix).



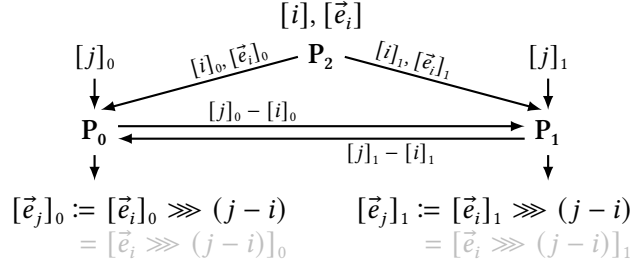


Figure 4.3: A  $(2+1)$ -party protocol for converting an additive sharing  $[j]$  of a scalar  $j \in [0 \dots N)$  into an additive sharing of the  $j$ th selection vector  $\vec{e}_j$  of length  $N$ .

#### 4.2.1 Scalar-to-selection vector share conversion

**Observation 4** suggests the following  $(2+1)$ -party protocol for converting an additive sharing of  $j$  into additive shares of the  $j$ th selection vector. Let  $N$  be given. In a preprocessing phase, a semi-trusted third party  $P_2$  chooses  $i \in [0 \dots N)$  uniformly and provides additive sharings  $[i]$  and  $[\vec{e}_i]$  to the first parties  $P_0$  and  $P_1$ , where  $\vec{e}_i$  is the  $i$ th selection vector of length  $N$ . Upon learning the sharing  $[j]$  in the online phase,  $P_0$  and  $P_1$  interactively reconstruct  $(j-i) \bmod N$  using [Equation \(4.2\)](#) and then they compute shares of  $\vec{e}_j$  via  $[\vec{e}_j] := [\vec{e}_i] \gg (j-i) \bmod N$ . A diagrammatic view of this  $(2+1)$ -party protocol is included as [Figure 4.3](#).

#### Share conversion in PIR

One case where producing a secret shared selection vector from a secret shared scalar, as described in [Section 4.2.1](#), is useful is in the context of PIR. Now, in a standard selection vector-based PIR scheme, the client performing the read already knows the index  $i$  to read from and can produce  $[\vec{e}_i]$  accordingly. However, when multiple parties in an MPC protocol want to perform a PIR query using selection vectors, it requires  $O(N)$  communication and computation to directly produce  $[\vec{e}_i]$  from shares of  $i$ . Instead,  $[j]$  and  $[\vec{e}_j]$  can be precomputed ahead of time, and the vector can then be rotated by  $i-j$  to produce  $[\vec{e}_i]$ . This requires only a single round of communication.

In this case, there exists an alternative to producing  $[\vec{e}_i]$  by rotating  $[\vec{e}_j]$  to the right by  $i-j$ . Instead the parties in the MPC protocol can rotate  $\vec{P}$  to the left by

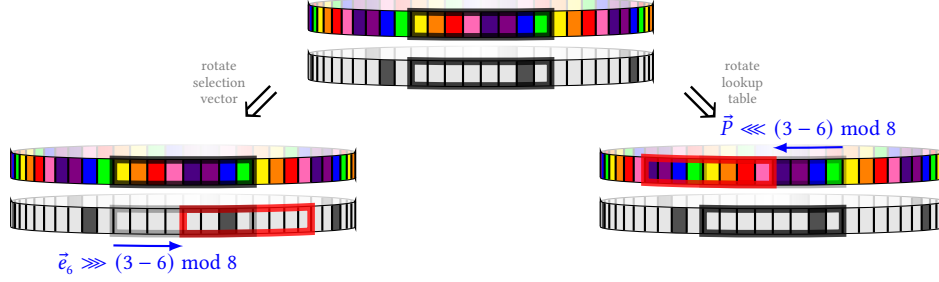


Figure 4.4: Equivalence between rotating selection vectors rightward versus databases leftward. In the diagram, the selection vector is  $\vec{e}_6$  and the desired record is  $P_3$  (the red element). The left subdiagram shows the outcome of rotating  $\vec{e}_6$  rightward to get  $\vec{e}_3$ ; the right subdiagram shows the outcome rotating  $\vec{P}$  leftward to move the red element into position 6.

the same distance. Taking inner products as before, this alternative is guaranteed to produce the same result because (i) commutativity implies inner products are invariant under cyclic reordering of summands, and (ii) left and right cyclic rotations are mutually inverse operations, so that

$$\begin{aligned}
 \langle \vec{e}_j, \vec{P} \rangle &= \langle \vec{e}_i \gg (j-i), \vec{P} \rangle \\
 &= \langle (\vec{e}_i \gg (j-i)) \ll (j-i), \vec{P} \ll (j-i) \rangle && \text{via (i)} \\
 &= \langle \vec{e}_i, \vec{P} \ll (j-i) \rangle. && \text{via (ii)}
 \end{aligned}$$

Figure 4.4 illustrates the equivalence between these two options. This equivalence is important for DPF-based PIR, because it is easier to rotate the database  $\vec{P}$  rather than a DPF output.

### 4.3 DPFs as parity-segment trees

The following observation about segment parities over point functions is obvious, yet it is sufficiently central to GROTTO as to warrant explication.

**Observation 5.** Fix  $i \in [0 \dots N)$  and let  $\vec{e}_i$  be the  $i$ th selection vector of length  $N$ . Then  $\text{parity}(\vec{e}_i[a \dots b]) = 1$  if and only if  $i \in [a \dots b)$ .

The following observation about the leaves in point functions is immediate.

**Observation 6.** *In a binary point function, the parity of the  $\lambda$ -bit vector held in the 1-leaf is 1 while the parity of the  $\lambda$ -bit vector held in each 0-leaf is 0.*

Composing [Observation 5](#) and [Observation 6](#) yields the following (relatively obvious) theorem, which serves as one of two lynchpins of the GROTTO construction presented in [Chapter 5](#).

**Theorem 5.** *Fix  $b \in \{0, 1\}$ . If  $P_{x,y}$  is a point function on  $\mathbb{Z}_N$  and  $B(P_{x,y})$  is its binary-tree representation, then an interior node  $X$  of  $B(P_{x,y})$  is a  $b$ -node if and only if the joint parity of the vectors stored in leaves descendant from  $X$  is  $b$ .*

When combined with [Theorem 5](#), [Observation 5](#) leads to the following implication.

**Corollary 1** (Point-function tree  $\rightarrow$  parity-segment tree). *Let  $P_i$  be the  $i$ th point function of length  $2^{n+k}$  and let  $B(P_i)$  be its binary-tree representation. Affixing to each node of  $B(P_i)$  the value 1 if it is a 1-node or a 1-leaf and 0 otherwise produces a parity-segment tree for  $\vec{e}_i$ .*

Meanwhile, from [Observation 2](#) we know that the joint parity of advice bits for 0-pairs is even (0) while for 1-pairs it is odd (1). In conjunction with [Observation 3](#), we get the following analogue of [Corollary 1](#).

**Corollary 2** (DPF tree  $\rightarrow$  point-function tree). *Let  $\llbracket \vec{e}_i \rrbracket = (\llbracket \vec{e}_i \rrbracket_0, \llbracket \vec{e}_i \rrbracket_1)$  be a binary DPF sharing of the  $i$ th point function of length  $2^{n+k}$  and let  $B(\llbracket \vec{e}_i \rrbracket_0)$  and  $B(\llbracket \vec{e}_i \rrbracket_1)$  be the corresponding BGI tree shares. Equating the value of each node of  $B(\llbracket \vec{e}_i \rrbracket_0)$  and  $B(\llbracket \vec{e}_i \rrbracket_1)$  with the advice bit stored within produces an XOR sharing of the binary-tree representation of  $\vec{e}_i$ .*

Finally, we note that all arithmetic operations in the prefix-parity algorithm are linear over  $\mathbb{Z}_2$ . The confluence of this fact with a transitive application of [Corollary 2](#) followed by [Corollary 1](#) makes the following “Fundamental Theorem of GROTTO” inescapable.

**Theorem 6** (Fundamental Theorem of GROTTO). *BGI shareholders can run the prefix-parity algorithm directly on their respective DPF shares to obtain XOR-sharings of arbitrary segment parities, at a cost of one half-PRG<sup>3</sup> evaluation per edge traversed.*

[Theorem 6](#) forms the basis of the GROTTO MPC system which is introduced in the following chapter.

---

<sup>3</sup>A *half-PRG* evaluation is an evaluation of  $G_\lambda$  in which only half of the output is required. For many PRGs, it is possible to compute only the required half at half the cost of a full, length-doubling evaluation.

## Chapter 5

# Grotto: MPC via (2, 2)-DPFs

This chapter introduces GROTTO, a system for efficient DPF-based MPC evaluation of non-linear functions. The method used by GROTTO fuses spline based approximation techniques from existing works on DPF- and DCF-based MPC [35, 60] with the efficient prefix-parity computation described in Chapter 4. One example where this system is valuable is when constructing private neural networks, where non-linear activation functions intersperse between linear layers. Early systems like SecureML [51] resorted to using “MPC-friendly” knockoffs of the activation functions used in non-MPC domains. In stark contrast, GROTTO enables efficient evaluation of non-linear functions where the accuracy is only limited by the precision of the fixed-point arithmetic being used.

This chapter introduces GROTTO, a framework and C++ library for space- and time-efficient  $(2 + 1)$ -party piecewise polynomial (i.e., spline) evaluation on secrets additively shared over  $\mathbb{Z}_2^n$ . GROTTO improves on the state-of-the-art approaches based on DCFs [35] and on DPFs [60, 62] in almost every metric, offering asymptotically superior communication and computation costs with the same or lower round complexity. At the heart of GROTTO is the novel observation about the relationship between the structure of Boyle–Gilboa–Ishai DPF trees and parity-segment trees, which we presented in Theorem 6. This allows GROTTO to do with a single lightweight DPF what state-of-the-art approaches require comparatively heavyweight DCFs to do. Our open-source GROTTO implementation supports evaluating dozens of useful functions out of the box, including trigonometric and hyperbolic functions (and their

inverses); various logarithms; roots, reciprocals, and reciprocal roots; sign testing and bit counting; and over two dozen of the most common (univariate) activation functions from the deep-learning literature.

### 5.0.1 The shoulders GROTTO stands upon

Our methods follow a line of research [11, 35, 41, 56, 60, 62] that uses *piecewise-polynomial functions* (or *splines*) to approximate and evaluate non-linear univariate functions on input additively shared secrets. The key innovation that GROTTO brings to this space is the introduction of a novel data structure called a *parity-segment tree* (and an associated *prefix-parity algorithm*), together with the observation that certain DPFs from the literature implicitly embed parity-segment trees within their internal structure. By leveraging this connection, we devise a space-, time-, and round-efficient way to obviously “select” the correct polynomial from a given piecewise representation.

Prior efforts in this space employ one of two competing strategies for this oblivious polynomial selection. The first strategy, introduced by Vadapalli, Bayatbabolghani, and Henry [60] for their Pirsona scheme, uses linear-sized DPFs with an exponential-cost *full-domain evaluation* procedure to evaluate reciprocal square roots and integer comparison. The follow-up scheme Pika [62] generalizes Pirsona’s approach to arbitrary functions and also adds machinery to thwart malicious dealers. This DPF-plus-full-domain-evaluation approach has low communication cost and concretely efficient running times for “short” inputs (say, 16–24 bits), but its exponential computation cost quickly grows untenable as inputs get longer; indeed, Wagh writes that “typical sizes for which [this approach] provides performance comparable to general purpose (sic) MPC are around 20–25 bits” [62; §3]. The recently proposed ORCA scheme of Jawalkar, Gupta, Basu, Chandran, Gupta, and Sharma [41] uses massive parallelism afforded by GPUs to partially reign in the exponential cost of full-domain evaluation.

The second strategy, exemplified by Gupta, Kumaraswamy, Gupta, and Chandran’s LLAMA scheme [35], uses *distributed comparison functions* (DCFs)—a DPF-adjacent primitive for efficient integer comparison—to avoid the need for costly full-domain evaluations. Swapping out DPFs in favour of DCFs dramatically im-

proves computational scaling at the expense of substantially higher communication costs (that increase rapidly as approximations become more granular).

Compared with its progenitors, GROTTO’s parity-segment approach offers concretely lower costs for “simple” functions and superior asymptotics as inputs get longer and approximations more elaborate. The upshot is that, while the respectively high computation and communication costs of the Pirsona and LLAMA frameworks severely curtail those schemes’ practically achievable approximation accuracy, the fidelity of GROTTO approximations is practically limited only by the difficulty of building good piecewise-polynomial approximations—indeed, as seen in [Table 5.2](#), GROTTO “approximations” are often *errorless* when viewed as fixed-point computations.

### 5.0.2 Threat model

The GROTTO protocols operate in the  $(2 + 1)$ -party computation setting described in [Section 2.5](#). As a result of working in this setting, the GROTTO protocols assume three pairwise non-colluding and semi-honest parties.

The research literature describes approaches based on *linear sketching* [[14, 21, 62](#)] or *secret-shared non-interactive proofs*, such as those used in Express [[25](#)] and in the Sabre system presented in [Chapter 7](#), that (combined with other standard techniques) could be used to relax the assumption that parties are semi-honest. Moreover, in contrast with general 3-party computation, the restricted role of the “server” makes  $(2 + 1)$ -party computation protocols particularly amenable to conversion into purely 2-party protocols:  $P_0$  and  $P_1$ , the two non-server parties, employ 2-party techniques to *emulate* the preprocessing steps of the third “server” party. We stress that this strategy works seamlessly because the online phase of a  $(2 + 1)$ -party protocol inherently (i) involves only the two non-server parties and (ii) is agnostic to the provenance of any correlated randomness those parties consume. Leveraging these ideas to port GROTTO to the maliciously secure 2-party setting is beyond the scope of this work; however, it remains an imperative and palpable direction for future work.

### 5.0.3 Roadmap

The remainder of the chapter proceeds as follows. [Section 5.1](#) revisits the Pirsona and LLAMA designs and elaborates on the innovations GROTTO brings to the table. [Section 5.3](#) synthesizes the ideas from preceding sections to create GROTTO, our framework and open-source software implementation of the new techniques, and then [Section 5.4](#) follows with a performance evaluation of GROTTO and a head-to-head comparison with prior art (namely, LLAMA [35]). [Section 5.5](#) wraps up the chapter with some concluding remarks.

## 5.1 Technical Overview

Before delving deeper into how GROTTO implements oblivious piecewise-polynomial evaluation, we take a step back and examine how the GROTTO approach relates to and differs from those taken by prior works. In particular, the relationship between the techniques used in GROTTO and those used in Pirsona [60], Pika [62], and LLAMA [35] at a “block-level” granularity. To a first approximation, all four schemes consist of two basic steps, namely (i) mapping the secret-shared input  $j$  to the appropriate polynomial and then subsequently (ii) obliviously evaluating that polynomial on input  $x = j$ .

The full-domain evaluation approach employed by Pirsona and Pika expands (linear-sized) DPF shares  $[\![\vec{e}_j]\!]$  into (exponential-sized) additive shares  $[\vec{e}_j]$  and then it proceeds to select the appropriate coefficient vector using inner products, precisely as described in [Section 2.8.2](#). (We stress that, despite its exponential cost, before GROTTO, full-domain evaluation was the most efficient way to complete this step with a single DPF.) For the second step, Pirsona and Pika each assume polynomials of degree  $d \leq 1$ , making polynomial evaluation a cookie-cutter application of standard techniques. The dealer distributes additive shares of the sign (which it learns while creating the DPF) so that sign correction, introduced in [Section 5.2](#), reduces to one additional application of plain-old Du-Atallah scalar multiplication.

LLAMA replaces DPFs encoding random selection vectors with DCFs encoding *the entire LUT* of the piecewise approximation. This approach bypasses the need for exponential-cost full-domain evaluation; instead, LLAMA shareholders need only



perform linear work for each “part” of the piecewise approximation. The tradeoff for this reduced workload is substantially larger communication cost,<sup>1</sup> with LLAMA DCFs often being several orders of magnitude larger than a DPF. LLAMA generally uses quadratic polynomials, which it then evaluates in the “obvious” way using multiple rounds of Du-Atallah. The way LLAMA encodes entire LUTs into the DCF shares obviates the need for sign correction.

In contrast, GROTTO leverages parity-segment trees and ABY2.0-like multiplication to effectively marry aspects of both designs, achieving LLAMA-like computation costs alongside Pirsona-like communication costs. We emphasize that this best-of-both-worlds design has practical implications that extend beyond shaving clock cycles and conserving bandwidth: It allows GROTTO to use piecewise approximations featuring more parts and higher-degree polynomials, paving the way for approximations with superior accuracy.

## 5.2 Function evaluation via PIR

If  $f: \mathbb{Z}_N \rightarrow \mathbb{Z}_N$  is some function and  $\vec{P}$  its truth table (that is,  $P_j := f(j)$  for all  $j \in \mathbb{Z}_N$ ), then using the DPF-based PIR scheme described in [Section 2.8.2](#) instantiates  $(2+1)$ -party oblivious evaluation of  $f(j)$  at the secret input  $j$ . When  $N$  is small (and  $f$  nonlinear), this procedure can perform well relative to evaluating  $f(j)$  directly using arithmetic (or Boolean) circuits [35, 60, 62]. We stress that the  $\mathbb{Z}_N$  elements may represent fixed-point numbers so that this is not limited to the oblivious evaluation of integer-valued functions.

As a potentially significant optimization, wherever  $f$  is *constant* within some interval,  $P_0$  and  $P_1$  can apply the distributive law to save some work in the inner product calculation. As an extreme example of this in action, suppose that  $f: \mathbb{Z}_N \rightarrow \mathbb{Z}_N$  is the step function defined by

$$f(j) := \begin{cases} A & \text{if } j \in [0 \dots 5), \text{ and} \\ B & \text{otherwise,} \end{cases}$$

<sup>1</sup>Specifically, the size of a LLAMA share scales as the product of the input size, the number of parts, and the degree of the constituent polynomials.

so that  $\vec{P} = (A, A, A, A, A, B, \dots, B)$ . Then, the inner product of  $[\vec{e}_j]_b = ([e_{j0}]_b, [e_{j1}]_b, \dots, [e_{jN}]_b)$  and  $\vec{P}$  has the very simple form

$$\begin{aligned} \langle [\vec{e}_j]_b, \vec{P} \rangle &= A \cdot ([e_{j0}]_b + [e_{j1}]_b + [e_{j2}]_b + [e_{j3}]_b + [e_{j4}]_b) \\ &\quad + B \cdot ([e_{j5}]_b + \dots + [e_{jN}]_b), \end{aligned}$$

which can be evaluated using  $N - 1$  additions and just two scalar multiplications. For comparison, a naïve evaluation would require  $N - 1$  additions and  $N$  scalar multiplications. (Furthermore, if, e.g.,  $B = 0$ , then the cost further shrinks to just 4 additions and a single scalar multiplication.)

### Function evaluation via binary selection vectors

So far, we have assumed that length- $N$  selection vectors  $\vec{e}_j$  are shared additively over  $\mathbb{Z}_N$ ; however, this is not a formal requirement. Indeed, because selection vectors consist solely of elements from  $\{0, 1\}$ , they can be shared more compactly as length- $N$  bitstrings (i.e., as length- $N$  vectors over  $\mathbb{Z}_2$ ). This shaves a factor  $\lceil \lg N \rceil$  from the size of  $\langle \vec{e}_j \rangle$  relative to that of  $[\vec{e}_j]$ , but not without introducing a minor technicality: Before they can evaluate the inner product between  $\langle \vec{e}_j \rangle$  and  $\vec{P}$ , the shareholders  $P_0$  and  $P_1$  must first *lift* each bit of  $\langle \vec{e}_j \rangle$  into an additive sharing over  $\mathbb{Z}_N$ . In general, such lifting is costly, requiring a round of interaction between  $P_0$  and  $P_1$  to ensure the resulting additive shares all have the correct *signs* in  $\mathbb{Z}_N$  (indeed, it is impossible to differentiate between  $\pm 1$  in  $\mathbb{Z}_2$ ).

Fortunately, the special form of selection vectors makes it possible for the shareholders to defer the latter interaction needed for sign correction to a *post-processing* step, to be performed only *after* evaluating the inner product.<sup>2</sup> In particular, for each of  $b = 0, 1$ , shareholder  $P_b$  lifts the  $i$ th bit  $\langle e_{ji} \rangle_b$  of  $\langle \vec{e}_j \rangle_b$  into an additive share over  $\mathbb{Z}_N$  via

$$[\pm e_{ji}]_b := \begin{cases} 0 & \text{if } \langle e_{ji} \rangle_b = 0, \text{ and} \\ (-1)^b & \text{otherwise,} \end{cases} \quad (5.1)$$

<sup>2</sup>Specifically, since  $\vec{e}_j$  consists entirely of 0s save for the 1 in position  $j$ , the initial lifting of  $\langle \vec{e}_j \rangle$  into  $\mathbb{Z}_N$  yields  $[\pm \vec{e}_j]$ ; that is, the requisite sign correction can occur at the granularity of the *entire vector*. For vectors with two or more non-zero entries, this would not be true.

so that  $[\pm e_{ji}]_0 + [\pm e_{ji}]_1 \in \{-1, 0, 1\}$ . Now, the inner product  $\langle [\pm \vec{e}_j], \vec{P} \rangle$  yields  $\pm P_j$ , a scalar that is correct up to sign. From here, there are a few options for how to implement the sign correction; we defer our discussion of those techniques to [Section 5.3.1](#).

The earlier optimization for when  $\vec{P}$  is constant over some interval ports nicely to the case where  $\vec{e}_j$  is shared bitwise as  $(\vec{e}_j)$ : The shareholders simply perform the required summation over  $\mathbb{Z}_2$  and then convert the resulting sums (i.e., not the individual addends) into additive shares over  $\mathbb{Z}_N$  using [Equation \(5.1\)](#). Notice that computing such sums of segments of  $(\vec{e}_j)$  is equivalent to computing the *parities* of the bitstrings corresponding to those segments.

As a concrete example, let us consider the evaluation of the step function  $f$  from the earlier example with vector length  $N = 8$  and input  $j = 4$ . Supposing  $\vec{e}_4$  is shared as  $(\vec{e}_4) = (1101\underline{1}010, 1101\underline{0}010)$ , shareholder  $P_0$  computes

$$\begin{aligned} (1 \oplus 1 \oplus 0 \oplus 1 \oplus \underline{1}, 0 \oplus 1 \oplus 0) &= (\text{parity}(1101\underline{1}), \text{parity}(010)) \\ &= (\underline{0}, 1), \end{aligned}$$

while shareholder  $P_1$  computes

$$\begin{aligned} (1 \oplus 1 \oplus 0 \oplus 1 \oplus \underline{0}, 0 \oplus 1 \oplus 0) &= (\text{parity}(1101\underline{0}), \text{parity}(010)) \\ &= (\underline{1}, 1). \end{aligned}$$

Upon lifting these values to  $\mathbb{Z}_N$ , the shareholders respectively hold vectors  $[-\vec{e}_4]_0 := (0, 1)$  and  $[-\vec{e}_4]_1 := (-1, -1)$ , from which they evaluate

$$\begin{aligned} \langle [-\vec{e}_4]_0, \vec{P} \rangle &= 0 \cdot a + 1 \cdot b \\ &= b \end{aligned}$$

and

$$\begin{aligned} \langle [-\vec{e}_4]_1, \vec{P} \rangle &= (-1) \cdot a + (-1) \cdot b \\ &= (-a) + (-b), \end{aligned}$$

and we find that  $b + ((-a) + (-b)) = -a$ , the negation of  $f(4)$ . A sign correction completes the process.

### Spline evaluation via selection vectors

Up until now, we have considered LUTs only for scalar-valued functions, yet the technique generalizes seamlessly to vector- or matrix-valued functions. As one useful application of this, we can evaluate functions  $\mathcal{F}: \mathbb{Z}_N \rightarrow (\mathbb{Z}_N)^{1 \times d}$  that output (vectors of coefficients defining) polynomials, including *piecewise-linear functions* and general *splines*. As a bonus, in light of the optimizations already discussed, such piecewise functions typically result in comparatively inexpensive inner product computations (i.e., requiring far fewer than  $N$  multiplications, since each “part” covers a non-trivial subinterval of the domain).

To see why this is useful, suppose we wish to approximate some highly nonlinear function  $f(x)$  that is prohibitively costly to evaluate exactly using arithmetic circuits. We can do so by constructing a piecewise-polynomial function  $\mathcal{F}$  such that, for all  $j \in \mathbb{Z}_N$ , the coefficients vector  $\langle a_d, \dots, a_1, a_0 \rangle \leftarrow \mathcal{F}(j)$  defines a good low-degree-polynomial approximation  $f'(x) = a_d x^d + \dots + a_1 x + a_0$  to  $f(x)$  in the vicinity of  $x = j$ . Given additive sharings of such a coefficient vector  $[\mathcal{F}(j)] = [f'(\cdot)]$  and of the input  $[j]$ , several well-known techniques can obviously compute  $[f'(j)]$ , thereby obtaining shares of a good approximation to  $f(j)$ . We describe two such techniques in Section 5.3.1, one based on Horner’s method [37] together with Du-Atallah multiplication [24] and the other on ABY2.0-style  $D$ -ary multiplication [54].

#### 5.2.1 Fractional-bit reduction for shared secrets

When evaluating polynomials, such as those in the previously discussed spline approximations, using fixed-point numbers, it quickly becomes necessary to reduce the fractional precision of the numbers being computed. In MPC settings, like GROTTO, reducing the fractional precision of a (2, 2)-additively shared fixed-point number  $[z]$  is similar to—albeit somewhat more tedious than—directly reducing that of  $z$ . Recall that in a two’s-complement encoding, the most-significant bit of  $z$  is a *sign bit* with  $\text{msb}(z) = 1$  if  $z$  is negative and  $\text{msb}(z) = 0$  if it is non-negative. Consequently, the most significant  $p$  bits of  $z \gg p$  are each “redundant” copies of the original sign bit (which is now the  $(p + 1)$ th-most-significant bit).

To reduce the number of fractional bits in  $[z] = ([z]_0, [z]_1)$ , each shareholder  $b$

computes

$$[\tilde{z}]_b := ([z]_b \gg p).$$

From here, there are two potential sources of errors, respectively manifesting in (i) the redundant sign bits and (ii) the least-significant bit of the resulting secret.

### Redundant sign bit errors

For errors in the redundant sign bits (i.e., the highest-order bits that were “shifted in”), there are three cases to consider:

**Case 1 ( $\text{msb}(z) = 0$ ;  $\text{msb}([z]_0) = \text{msb}([z]_1) = 1$ ):** Here  $[z]_0 + [z]_1$  overflows (carries out from the most-significant bit) so that reconstructing  $z$  entails an *implicit* reduction modulo  $2^{64}$ . Furthermore, each of the  $p$  most-significant bits of  $[\tilde{z}]_b$  are set; hence, in the sum  $[\tilde{z}]_0 + [\tilde{z}]_1$ , the carry-out from the  $(p + 1)$ th-most-significant bit induces a carry chain that leaves the  $p$  leftmost bits errantly set. Consequently,

$$[\tilde{z}]_0 + [\tilde{z}]_1 + 2^{64-p} \equiv z \pmod{2^{64}}.$$

**Case 2 ( $\text{msb}(z) = 1$ ;  $\text{msb}([z]_0) = \text{msb}([z]_1) = 0$ ):** This is similar to the first case, except now signs are flipped so that

$$[\tilde{z}]_0 + [\tilde{z}]_1 - 2^{64-p} \equiv z \pmod{2^{64}}.$$

**Case 3 ( $\text{msb}(z) = \text{msb}([z]_0)$  or  $\text{msb}(z) = \text{msb}([z]_1)$ ):** It is easy to check that

$$[\tilde{z}]_0 + [\tilde{z}]_1 \equiv z \pmod{2^{64}}$$

always holds in this case.

The first two cases require an additional correction, wherein the shareholders conditionally (and obviously) add  $[\pm 2^{64-p}]$  to  $[\tilde{z}]$  to get  $[z \gg p]$ . There exist a multitude of options for how to implement this conditional correction; however, they all require one or more rounds of interaction. Computationally, the “best” case occurs when  $\text{msb}(z)$  is known (say, because application logic allows its deduction)

so that what correction to apply depends solely on *either*  $\text{msb}([z]_0) \wedge \text{msb}([z]_1)$  or  $\neg \text{msb}([z]_0) \wedge \neg \text{msb}([z]_1)$ . For the general case with  $B$ -bit integers where  $\text{msb}(z)$  is *not* known, we can always use

$$[C] := 2^{B-p} \cdot (Z_0 \cdot Z_1 + (Z_0 + Z_1 - 2 \cdot Z_0 \cdot Z_1 - 1) \cdot [Z]), \quad (5.2)$$

where  $C$  is the necessary correction value,  $Z_b = \text{msb}([z]_b)$  for  $b = 0, 1$ , and  $Z = \text{msb}(z)$ .

*Proof.* To prove the correctness of Equation (5.2), we consider the three possible cases discussed previously.

**Case 1 ( $\text{msb}(z) = 0$ ;  $\text{msb}([z]_0) = \text{msb}([z]_1) = 1$ ):** Inserting the given values into Equation (5.2) produces

$$\begin{aligned} [C] &:= 2^{B-p} \cdot ([1] \cdot [1] + (1 + 1 - 2 \cdot 1 \cdot 1 - 1) \cdot [0]) \\ &= 2^{B-p} \cdot ([1] - [0]) \\ &= [2^{B-p}] \end{aligned}$$

As discussed previously, when reducing  $[z]$  by  $p$  bits, this case results in the  $p$  leftmost bits being set incorrectly. Adding  $[2^{B-p}]$  to  $[\tilde{z}]$  corrects this, so that the result is  $[z \gg p]$ .

**Case 2 ( $\text{msb}(z) = 1$ ;  $\text{msb}([z]_0) = \text{msb}([z]_1) = 0$ ):** Inserting the given values into Equation (5.2) produces

$$\begin{aligned} [C] &:= 2^{B-p} \cdot (0 \cdot 0 + (0 + 0 - 2 \cdot 0 \cdot 0 - 1) \cdot [1]) \\ &= 2^{B-p} \cdot [-1] \\ &= [-2^{B-p}] \end{aligned}$$

As previously observed, in this case,  $[\tilde{z}]$  has  $p$  leading bits which are errantly cleared.  $C = -2^{B-p}$  is a number in which the  $p$  leading bits are all set and all other bits are cleared. Therefore, adding  $[C]$  to  $[\tilde{z}]$  sets the  $p$  leading bits to produce  $[z \gg p]$ .

**Case 3 ( $\text{msb}(z) = \text{msb}([z]_0)$  or  $\text{msb}(z) = \text{msb}([z]_1)$ ):** Since  $Z_0$  and  $Z_1$  are used symmetrically in Equation (5.2), we can assume, without loss of generality, that  $\text{msb}(z) = \text{msb}([z]_0)$ . This leads to the following result

$$[C] := 2^{B-p} \cdot (Z \cdot Z_1 + (Z + Z_1 - 2 \cdot Z \cdot Z_1 - 1) \cdot [Z])$$

Clearly, if  $Z = Z_0 = 0$ , then  $[C] = [0]$ . Similarly, if  $Z = Z_0 = 1$ , then the result is

$$\begin{aligned} [C] &:= 2^{B-p} \cdot (1 \cdot Z_1 + (1 + Z_1 - 2 \cdot 1 \cdot Z_1 - 1) \cdot [1]) \\ &= 2^{B-p} \cdot (Z_1 + (1 - Z_1 - 1) \cdot [1]) \\ &= 2^{B-p} \cdot (Z_1 - [Z_1]) \\ &= [0] \end{aligned}$$

For both values of  $Z = Z_0$ , the result is  $[C] = [0]$ . Since  $[\tilde{z}]_0 + [\tilde{z}]_1 \equiv z \pmod{2^{64}}$  in this case, adding  $[C] = [0]$  to  $[\tilde{z}]$  will produce shares of  $z \gg p$  with correct leading bits.

□

Mohassel and Zhang prove [51; Appendix B] that each of Cases 1 and 2 only occur with probability negligible in the number of “extra” integer bits; thus, if program logic suffices to prove that integer parts are sufficiently small,  $P_0$  and  $P_1$  can forgo explicit corrections and still get the correct result with very high probability.

**Redundant sign bit example** To illustrate why such “corrections” are needed, we consider the problem of resetting the number of fractional bits in the area of a circle. Let  $A = 0x0004e8a270000000$ , as computed in Section 2.2, and consider the (2, 2)-additive sharing of  $A$  via

$$[A]_0 = 0x80014bf69ed29a6b$$

and

$$[A]_1 = 0x80039cabd12d6595.$$

Notice that  $\text{msb}([A]_0) = \text{msb}([A]_1) = 1$  whereas  $\text{msb}(A) = 0$ , yet  $[A]_0 + [A]_1 = A$  over  $\mathbb{Z}_{2^{64}}$ . Then

$$(A \gg 32) = 0x \overbrace{00000000}^{\text{redundant sign bits}} \overbrace{00004e8a2}^{\text{fractional part}},$$

original sign bit + integer part

while

$$[\tilde{A}]_0 = ([A]_0 \gg 32) = 0x \underbrace{ffffff}_{\text{redundant sign bits}} \overbrace{80014bf6}^{\text{shifted share}}$$

and

$$[\tilde{A}]_1 = ([A]_1 \gg 32) = 0x \underbrace{ffffff}_{\text{redundant sign bits}} \overbrace{80039cab}^{\text{shifted share}},$$

so that

$$\begin{aligned} ([\tilde{A}]_0 + [\tilde{A}]_1) + 2^{64-32} &\equiv (0xffffffff80014bf6 \\ &\quad + 0xffffffff80039cab) \\ &\quad + 0x0000000100000000 \\ &\equiv 0x \textcolor{red}{ffffff} 0004e8a1 \\ &\quad + 0x0000000100000000 \\ &\equiv 0x000000000004e8a1 \pmod{2^{64}}. \end{aligned}$$

Now, we can observe that  $([\tilde{A}]_0 + [\tilde{A}]_1) + 2^{64-32} = (A \gg 32) - 1$ . In order to get the exact result  $A \gg 32$ , we must correct the least-significant bit error in this precision reduction.

### Least-significant bit errors

The second source of error occurs when the  $p$  low-order bits that get shifted out of  $[z]$  would have induced a carry-in to new least-significant bit—an event that occurs with probability 0.5. When this event occurs, we find that  $[\pm 2^{64-p}] + [\tilde{z}] = [(z \gg p) - 1]$  is the *next smallest representable number* from the one we desire; that is, our answer undershoots the correct answer by  $2^{-(q+1)}$  in expectation (and up to  $2^{-q}$  in the worst case), where  $q$  is the number of bits of fractional precision after the precision reduction.



To correct this error, it suffices to compute and add shares of the carry-in that *would have* resulted from the  $p$  shifted-out bits. We can compute this carry-out either using GROTTO (with a fresh DPF) or using a DCF. As the sole fractional-precision reduction in a GROTTO invocation occurs only *after* the sign-corrected polynomial evaluation, both options require at least one additional round of communication; thus, in order to minimize the impact on round complexity, our implementation use a DCF that adds only a single round.<sup>3</sup> We stress, however, that the general efficiency of GROTTO permits evaluations that use highly granular LUTs and fixed-points with many fractional bits, allowing even *uncorrected* GROTTO evaluations to provide very good approximations. Our experiments in [Section 5.4](#) indicate the cost and accuracy of GROTTO evaluations both with and without this correction.

**Least-significant bit example** To demonstrate a least-significant bit error and how it is corrected, we continue with the problem of resetting the number of fractional bits in the area of a circle, which we began in [Section 5.2.1](#). In this example, the value  $A = 0x0004e8a270000000$  is divided into the (2, 2)-additive sharing

$$[A]_0 = 0x80014bf69ed29a6b$$

and

$$[A]_1 = 0x80039cabd12d6595.$$

[Section 5.2.1](#) showed how the redundant sign-bit errors that appear in the precision reduction can be corrected. Let  $[\bar{A}]_0$  and  $[\bar{A}]_1$  denote the reduced shares after the correction for the most significant bits has been applied. However, even after these corrections are applied,  $[\bar{A}]_0 + [\bar{A}]_1 = (A \gg 32) - 1$ , not  $A \gg 32$ . This is due to a least-significant bit error.

Let the low-order 32 bits of  $[A]_0$  and  $[A]_1$  be denoted as  $[L]_0 = 0x9ed29a6b$  and  $[L]_1 = 0xd12d6595$  respectively. We can take the sum of these two values to see that

$$\begin{aligned} [L]_0 + [L]_1 &= 0x9ed29a6b + 0xd12d6595 \\ &= 0x170000000 \end{aligned}$$

---

<sup>3</sup>Note that the DCF shares for a single 64-bit comparison is only about  $\lambda \lg \lambda + 64^2$  bits larger than the corresponding GROTTO DPF; the tradeoff for this extra size relative to using a DPF is one fewer communication round.

$$0x9ed29a6b + 0xd12d6595 = 0x170000000.$$

This shows that there is a carry out from the low-order 32 bits of the shares of  $A$ . The least-significant bit error results because these low-order bits are omitted in  $[\tilde{A}]$  and  $[\bar{A}]$ .

To correct for this error, we consider  $[L]_0$  and  $[L]_1$ . We then append a zero bit as the most significant bit of both shares, so that they can be interpreted as 33-bit shares. Now, determining whether  $[L]_0 + [L]_1$  has a carry out bit over 32-bit numbers is equivalent to determining if the MSB of  $[L]_0 + [L]_1$  is set over 33-bit numbers. As stated previously, this is a problem that can be easily answered using GROTTO or a DCF. The resulting shares of the carry out bit  $[L \gg 32]$  can then be added to  $[\bar{A}]$ , correcting the LSB value. This produces an exact precision reduction of  $[A]$ .

### 5.3 The GROTTO framework

We now have all the fundamental building blocks in place. This section describes how we integrated these building blocks to arrive at GROTTO, our framework and C++ library for space- and time-efficient  $(2+1)$ -party evaluation of piecewise-polynomial functions (or splines) on  $(2, 2)$ -additively shared inputs.

#### The premise

$P_0$  and  $P_1$  wish to obliviously evaluate some non-linear function  $f: \mathbb{R} \rightarrow \mathbb{R}$  on input some  $(2, 2)$ -additively shared fixed-point value  $[x]$ ; that is, they wish to compute  $[f(x)]$  from  $[x]$ . We assume that  $f$  is well-approximated by the piecewise-polynomial function described in  $\mathcal{F} := (\vec{B}, \vec{P})$  and that  $P_0$  and  $P_1$  each hold  $\mathcal{F}$  in plaintext. Here  $\vec{B}$  is the ordered list of endpoints for the “pieces” of the approximation and  $\vec{P}$  is the correspondingly ordered list of (vectors of coefficients defining) polynomials for approximating within those pieces.

#### Preprocessing phase

In a preprocessing phase, some benevolent third-party ( $P_2$ ) samples an  $([i], \llbracket \vec{e}_i \rrbracket)$  pair alongside “Beaver triple-like” values in support of the eventual sign-corrected

polynomial evaluation (see [Section 5.3.1](#)), and then it distributes the shares and Beaver triple-like values to  $P_0$  and  $P_1$  and exits the scene.<sup>4</sup>

### Online phase

Upon learning  $[x]$  in the online phase,  $P_0$  and  $P_1$  use a reconstructed  $x - i$  to cyclically shift each endpoint in  $\vec{B}$  to the left. Here they are running the DPF-based PIR protocol introduced in [Section 2.8.2](#) along with the technique of rotating  $\vec{B}$  discussed in [Section 4.2](#). We emphasize that shifting  $\vec{B}$  instead of  $\vec{e}_i$  obviates the need for  $P_0$  and  $P_1$  to evaluate the DPF at every  $i \in [0 \dots N)$ , a procedure that may be prohibitively costly—or perhaps even computationally infeasible—when  $N$  is large [62].

Both parties then run the prefix-parity algorithm on their respective shares of  $[\vec{e}_i]$  to find XOR-shared prefix parities for each of the above-rotated endpoints, and then they use these XOR-shared prefix parities to construct XOR-shares of the vector of segment parities corresponding to pieces in  $\mathcal{F}$ . Specifically, the resulting shares reconstruct to the selection vector indicating which polynomial reflected in  $\vec{P}$  provides a good approximation to  $f$  on input  $x$ . From here, the two parties use this vector of parities to obviously fetch *additive* shares of (plus-or-minus) the appropriate coefficients vector from  $\vec{P}$ , using the PIR-like process from [Section 2.8.1](#).

Finally, the parties use the aforementioned Beaver triple-like values to compute sign-corrected evaluations on input  $[x]$  of whatever polynomial  $f'$  they obviously fetched in the preceding step, yielding additive shares of (a good approximation to) the desired evaluation  $f(x)$ .<sup>5</sup>

#### 5.3.1 Sign-corrected polynomial evaluation

GROTTO could use any of a number of known techniques for oblivious polynomial evaluation. Our implementation supports two such techniques, namely either (i) Horner’s method together with Du-Atallah multiplication or (ii) ABY2.0-style  $D$ -ary multiplication.

<sup>4</sup>Depending on the method being used to evaluate the polynomial and correct the output sign, this preprocessed value will differ as discussed in [Section 5.3.1](#).

<sup>5</sup>The details the evaluation are intentionally left vague here, because these particulars are dependent on the evaluation and sign-correction method employed.

### Horner's method

Horner's method is a technique for evaluating polynomials efficiently. To evaluate the degree- $d$  polynomial  $a(x) := a_d \cdot x^d + a_{d-1} \cdot x^{d-1} + \dots + a_1 \cdot x + a_0$ , Horner's method simply expresses it in the form

$$a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot (a_3 + \dots + x \cdot (a_{d-1} + x \cdot a_d) \dots))),$$

thereby allowing its evaluation via an interleaved sequence of  $n$  additions and  $n$  multiplications.

For example, to evaluate the quadratic  $f(x) = a_2 \cdot x^2 + a_1 \cdot x + a_0$ , in which the coefficients  $a_i$  and indeterminate  $x$  are each fixed-point numbers with  $p$  fractional bits, on input  $x = j$ , Horner's method evaluates the expression

$$f(j) = (((((a_2 \cdot j) \gg p) + a_1) \cdot j) \gg p) + a_0.$$

The arithmetic right-shifts following every multiplication ensure the operands to each addition and the final output all have  $p$  fractional bits. In practice, the arithmetic right-shift to reduce the fractional precision does not need to be performed after every multiplication. The results will still be correct, within the constraints of the available bits for the integer and the fractional part of the number, as long as all additions are performed between fixed-point number with the same precision.

When this approach is instantiated using Du-Atallah multiplication, evaluating a degree- $d$  polynomial requires a minimum of  $d + 1$  rounds of interaction—specifically,  $d$  rounds for the  $d$  multiplications plus one final round that merely sign-corrects the penultimate answer. Interleaved within these rounds of multiplication will be some number of precision reduction steps. A precision reduction is inserted when the next multiplication would result in the intermediate value having too few integer bits. Each of these precision reductions adds a minimum of one round of communication if the sign of the intermediate value is known. If the sign is not already known, secret shares of the sign bit needs to be extracted and then two rounds of communication are required to complete the sign reduction. [Algorithm 1](#) shows pseudocode for Horner's method-based evaluation in which the precision is reduced whenever the intermediate value's precision exceeds a given  $p_{\max}$ . The details of this precision reduction are omitted as the method used is dependent on context.

**Algorithm 1** Du-Atallah Horner's Method

---

**Require:** Input precision  $p_x$   
**Require:** Coefficient precision  $p_c$   
**Require:** Maximum precision  $p_{\max}$   
**Require:** Fixed-point input  $x$   
**Require:** Fixed-point coefficients  $c_0, c_1, \dots, c_d$  ( $d \in \mathbb{N}$ ) for polynomial  $f(x) = c_d \cdot x^d + c_{d-1} \cdot x^{d-1} + \dots + c_1 \cdot x + c_0$   
 $[t] = [c_d]$   
 $p_{\text{current}} = p_c$   
**for**  $i = d, d-1, \dots, 1$  **do**  
    From  $P_2$  receive Beaver Triple  $B_i$   
    Compute  $[t] = [t] \cdot [x]$  using  $B_i$   
     $p_{\text{current}} = p_{\text{current}} + p_x$   
    **if**  $p_{\text{current}} \geq p_{\max}$  **then**  
        Reduce  $[t]$  to precision  $p_c$   
    **end if**  
     $[t] = [t] + ([c_{i-1}] \ll (p_{\text{current}} - p_c))$   
**end for**  
Output  $[t]$

---

**Sign-correcting the answer**

Implementing the sign-correction is pleasantly easy: Let  $\langle \tilde{e} \rangle = (\langle \tilde{e} \rangle_0, \langle \tilde{e} \rangle_1)$  be the XOR-shared vector of segment parities, let  $U_0$  and  $U_1$  respectively denote the sum over  $\mathbb{Z}_N$  of all parities in  $\langle \tilde{e} \rangle_0$  and  $\langle \tilde{e} \rangle_1$ , and set  $u := U_0 - U_1$ . By construction, we have  $u = \pm 1$  and, moreover, if  $f'$  is the (uncorrected) polynomial that  $\langle \tilde{e} \rangle$  selects from  $\vec{P}$ , then  $u$  has the “matching” sign. It, therefore, follows that

$$f(j) \approx u \cdot f'(j),$$

which  $P_0$  and  $P_1$  can easily compute a sharing of using one final Du-Atallah multiplication between the sharings  $[\pm f'(j)]$  and  $[u] = (U_0, -U_1)$ .

For example, consider the example from [Section 5.2](#), in which the selection vector parity shares  $\langle \vec{e}_4 \rangle_0 := (0, 1)$  and  $\langle \vec{e}_4 \rangle_1 := (1, 1)$  are lifted to the  $\mathbb{Z}_N$  selection vector shares  $[-\vec{e}_4]_0 := (0, 1)$  and  $[-\vec{e}_4]_1 := (-1, -1)$ . As a result, the inner products to retrieve values from  $\vec{P}$  yields,  $[r]_0 = \langle [-\vec{e}_4]_0, \vec{P} \rangle = b$  and  $[r]_1 = \langle [-\vec{e}_4]_1, \vec{P} \rangle = (-a) + (-b)$ . Thus,  $r = -a$ , which is the negation of the desired result  $f(4)$ . We can

then perform the sign correction. First, we use the same selection vector parity shares to retrieve a value from a table in which all entries have the value 1. Simplified, this produces sign correction shares  $[s]_0 = 0 + 1 = 1$  and  $[s]_1 = (-1) + (-1) = -2$ . Thus,  $s = -1$ , which is the negative because  $r = -a$  is the negation of the proper result. Then, shares of the correct answer  $f(4) = a = s \cdot r$  can be computed in MPC, either as a standalone Du-Atallah multiplication or as part of the ABY2.0-style multiplication discussed next.

### ABY2.0-style multiplication

The approach based on Horner’s method incurs low precomputation costs in exchange for a relatively large (degree-dependent) round complexity. As an alternative that avoids this blowup in round complexity, GROTTO also supports polynomial evaluation based on the single-round  $D$ -ary multiplication technique used by ABY2.0 [54], which is described in Section 2.5.3. This technique has a noticeably higher pre-processing cost, but it can be substantially more performant in instances where each round of communication incurs Internet round-trip latency.

Our use of ABY2.0-style multiplication follows the original expositions of Patra, Schneider, Suresh, and Yalame [54; §3.1.4] rather faithfully, save for two important optimizations that we introduce specifically to facilitate efficient fixed-point polynomial evaluation. First, because we desire only the final output of a polynomial evaluation (i.e., we are not interested in evaluating the individual monomials), we are able to merge several  $P_2$  terms to noticeably reduce overall precomputation size relative to a naïve application of ABY2.0-style multiplication to the individual monomials. Second, to prevent integer overflows and the need to reduce the fractional bits in intermediate values, we “lift” the coefficients  $a_i$  and indeterminate  $x$  to a larger ring (namely, to  $\mathbb{Z}_{2^{n+k+m}}$  for some  $m \in \mathbb{N}$ ). Once the polynomial has been evaluated in this larger ring, we project the result back into  $\mathbb{Z}_{2^{n+k}}$ . While it would be possible to support arbitrarily large integer parts with this approach, our implementation in GROTTO seeks only to support evaluations that would also succeed using Horner’s method (see Section 5.3.1); for instance, if  $x$  is a 64-bit integer with 16 fractional bits, then each intermediate value in the Horner evaluation has 32 fractional bits and, thus, integer parts comprising at most  $31 = 64 - 32 - 1$  bits. Therefore, for polynomials of

degree  $d$ , we desire a ring large enough to encode  $16(d+3)$  bits—31 integer bits, 1 sign bit, and  $16(d+1)$  fractional bits—such as  $\mathbb{Z}_{2^{n+k+m}}$  for any  $m \geq \max(16(d+3) - n - k, 0)$ . For optimal performance on 64-bit CPUs, our implementation uses the smallest such  $m$  for which  $n + k + m$  is a multiple of 64.

### Lifting the coefficients

Lifting the  $a_i$  is trivial: Since  $P_0$  and  $P_1$  hold the LUT  $\vec{P}$  in plaintext, they can lift each coefficient “for free” ahead of the PIR step. At the same time, they adjust the number of fractional bits in each coefficient (by left-shifting in zeros) so that every monomial  $a_i \cdot x^i$  will use exactly  $(d+1) \cdot p$  fractional bits; that is, they replace each  $a_i$  by  $\bar{a}_i := (a_i \ll (d-i) \cdot p) \parallel 0^{i \cdot p} \in \mathbb{Z}_{2^{n+k+m}}$ . The latter fractional-precision adjustments ensure that the decimal points in intermediate values of the polynomial evaluation “line up”, allowing them to be added or subtracted non-interactively.

### Lifting the indeterminate

Lifting the indeterminate is more cumbersome, as  $P_0$  and  $P_1$  hold only a sharing  $[j]$  of the input. The main observation behind our approach is that lifting  $[j]$  is actually trivial—*provided we are not fussy about the number of fractional bits in the lifted result*. Specifically, to lift  $[j]$  from  $\mathbb{Z}_{2^{n+k}}$  into  $\mathbb{Z}_{2^{n+k+m}}$ ,  $P_0$  and  $P_1$  each simply append  $0^m$  to the binary representations of their respective shares (and switch from reducing modulo  $2^{n+k}$  to reducing modulo  $2^{n+k+m}$ ) so that

$$[j]_0 \parallel 0^m + [j]_1 \parallel 0^m = j \parallel 0^m \in \mathbb{Z}_{2^{n+k+m}},$$

which is the correct  $j$ , only with  $p + m$  instead of  $p$  fractional bits. From here, an interactive fractional precision reduction (see [Section 5.2.1](#)) suffices to “reset” the number of fractional bits back to the desired  $p$ .

Only one question remains unanswered: How do  $P_0$  and  $P_1$  determine  $[\text{msb}(j)]$ , which is needed in [Equation \(5.2\)](#), from  $[j]$ ? For this, we look inward, noting that the msb function is just a piecewise-constant (indeed, piecewise-Boolean) function comprising two parts; thus, the parties can calculate  $[\pm \text{msb}(j)]$  using *the same DPF shares* as they are using to fetch  $f'$  from  $\vec{P}$ . Because the “polynomials” for the msb function are constant integers, there is no reason to lift them to the larger ring (i.e.,

there is no “chicken and egg” situation). Then, when subsequently computing the correction term via Equation (5.2), we use the fact that  $\text{msb}(j) \in \{0, 1\}$  so that

$$(\pm \text{msb}(j))^2 = (-\text{msb}(j))^2 = \text{msb}(j)^2 = \text{msb}(j)$$

holds for all  $j$ .

### Putting it all together

Instantiating GROTTO with ABY2.0-style multiplication yields a three-round protocol for approximations via polynomials of any degree:

**Round 1:**  $P_0$  and  $P_1$  reconstruct  $j - i \bmod 2^{n+k}$  and use a Du-Atallah multiplication to compute the sharing  $[Z_0 \cdot Z_1]$  with  $Z_b := \text{msb}([j]_b)$  for  $b = 0, 1$  (cf. Equation (5.2)). Note that  $P_0$  and  $P_1$  respectively know  $Z_0$  and  $Z_1$  in plaintext, so the plaintext variant of Du-Atallah multiplication can be used.

Before proceeding to the next round, each party lifts (and precision-adjusts) the coefficients comprising  $\vec{P}$  into  $\mathbb{Z}_{2^{n+k+m}}$  and then runs the prefix-parity algorithm to fetch its shares of  $[\pm \bar{a}_i]$  and  $[\pm Z]$  for  $Z := \text{msb}(j)$ .

**Round 2:**  $P_0$  and  $P_1$  use a ternary ABY2.0-style multiplication to compute the sharing  $[(Z_0 \cdot Z_1) \cdot (\pm Z)^2]$  from  $[Z_0 \cdot Z_1]$  and  $[\pm Z]$ , and then they use it to (from this point on, non-interactively) compute  $[\pm 2^{n+k}]$  using Equation (5.2).

Before proceeding to the next round, the parties use  $[\pm 2^{n+k}]$  as the correction term to lift the shares of the indeterminate  $x \in \mathbb{Z}_{2^{n+k}}$  into shares of  $\bar{x} \in \mathbb{Z}_{2^{n+k+m}}$ .

**Round 3:**  $P_0$  and  $P_1$  use a  $(d + 2)$ -ary ABY2.0-style multiplication over sharings  $[\bar{a}_0], \dots, [\bar{a}_d], [\bar{x}]$ , and  $[u]$  (which they compute the same way as they would in Horner’s method) to evaluate  $[f'(x)] = [u \cdot (a_d \cdot x^d + \dots + a_1 \cdot x + a_0)]$ .

**Round 4:**  $P_0$  and  $P_1$  use a DCF to obliviously compute—and add to their respective shares—the carry-in to the least-significant bit of  $[f'(x)]$ .

In cases where the coefficients and  $x$  need not be lifted to avoid overflow, we can skip **Round 2** above, resulting in a somewhat simpler two-round protocol. We



include our precise formulae for ABY2.0-style sign-corrected polynomial evaluation and detailed derivations thereof as [Appendix B](#).

In cases where the coefficients and  $x$  need not be lifted to avoid overflow, we can skip **Round 2** above, resulting in a somewhat simpler protocol with one less round. Likewise, when either the sign-corrected polynomial evaluation does not include a precision reduction or the result small bias is palatable, the last round can be omitted. We include our precise formulae for ABY2.0-style sign-corrected polynomial evaluation and detailed derivations thereof as [Appendix B](#).<sup>6</sup>

## 5.4 Implementation & evaluation

To empirically evaluate the performance of our approach, we implemented GROTTO as a C++ library. Our implementation uses `dpf++` [36] for (2, 2)-DPFs, the GNU multiprecision arithmetic library (GMP) v6.2.1 [34] for multi-limb arithmetic in our ABY2.0-style multiplication, and the C++ version of ALGLIB 3.19.0 [10] for curve fitting in our LUT-generation code.

In addition to implementing the prefix-parity algorithm and associated (2 + 1)-party protocols, our implementation comes equipped with *scores* of “gadgets” (i.e., LUTs and associated machinery) for evaluating common functions, including trigonometric and hyperbolic functions (and their inverses); various logarithms; roots, reciprocals, and reciprocal roots; sign testing and bit counting; and over two dozen of the most common (univariate) activation functions from the deep-learning literature. We also include utilities for generating additional LUTs from arbitrary functions  $f: \mathbb{R} \rightarrow \mathbb{R}$  given as a blackbox.<sup>7</sup>

---

<sup>6</sup>In fact, [Appendix B](#) includes derivations for both one- and two-round sign-corrected evaluations (for constant, linear, quadratic, and cubic polynomials). The two-round variants are similar to their one-round counterparts, as described above, except they apply the sign correction as a post-processing step (similar to with Horner’s method). This reduces the number of Beaver-like terms that  $P_2$  must send at the expense of one additional communication round; crucially, though, it still decouples the round complexity of polynomial evaluation from the degree of the polynomial under consideration.

<sup>7</sup>Simultaneously efficient, accurate, and fully-automated LUT generation is impossible given only blackbox access to  $f$ ; to work around this, our LUT-generation utility allows the user to provide some “hints” that effectively transform the problem into that of “graybox” LUT generation.

Table 5.1: Summary of selected gadgets with out-of-the-box support in Grotto, assuming 64-bit fixed-point arithmetic with 16 fractional bits.

Gadget	Descriptive name	Formula	Degree	# parts <sup>†</sup>	Max error <sup>‡</sup>	RMSE <sup>‡</sup>	Expected half-PRGs <sup>‡</sup>
cos	cosine <sup>†</sup>	$\cos(x)$	3	64	$3.0e-8$	$1.6e-8$	$586 \pm 3$
sin	sine <sup>†</sup>	$\sin(x)$	3	63	$3.0e-8$	$1.7e-8$	$580 \pm 2$
tan	tangent <sup>†</sup>	$\sin(x)/\cos(x)$	3	608	$4.6e-4^{\ddagger}$	$5.8e-5^{\ddagger}$	$489 \pm 3$
csc	cosecant <sup>†</sup>	$1/\sin(x)$	3	595	$4.6e-4^{\ddagger}$	$5.8e-5^{\ddagger}$	$492 \pm 3$
sec	secant <sup>†</sup>	$1/\cos(x)$	3	358	$4.6e-4^{\ddagger}$	$5.5e-5^{\ddagger}$	$496 \pm 3$
cot	cotangent <sup>†</sup>	$1/\tan(x)$	3	366	$2.9e-4^{\ddagger}$	$3.4e-5^{\ddagger}$	$486 \pm 3$
arccos	arc cosine	$\cos^{-1}(x)$	3	31	$6.8e-5$	$1.1e-5$	$245 \pm 2$
arcsin	arc sine	$\sin^{-1}(x)$	3	31	$7.9e-5$	$1.3e-5$	$244 \pm 2$
arctan	arc tangent	$\tan^{-1}(x)$	3	179	$1.6e-7$	$2.4e-8$	$2333 \pm 4$
arcsc	arc cosecant	$\csc^{-1}(x)$	3	55	$4.4e-4^{\ddagger}$	$3.9e-5^{\ddagger}$	$744 \pm 3$
arcsec	arc secant	$\sec^{-1}(x)$	3	58	$4.4e-4^{\ddagger}$	$3.5e-5^{\ddagger}$	$755 \pm 4$
arccot	arc cotangent	$\cot^{-1}(x)$	3	41	$1.5e-5^{\ddagger}$	$7.2e-6^{\ddagger}$	$684 \pm 3$
arcosh	area hyperbolic cosine	$\ln(x + \sqrt{x^2 - 1})$	3	439	$2.4e-7$	$8.7e-8$	$11129 \pm 4$
arsinh	area hyperbolic sine	$\ln(x + \sqrt{x^2 + 1})$	3	753	$2.4e-7$	$8.9e-8$	$22166 \pm 4$
artanh	area hyperbolic tangent	$\frac{1}{2} \ln((1+x)/(1-x))$	3	54	$1.5e-7$	$9.0e-8$	$299 \pm 2$
ln	natural logarithm	$\ln(x)$	3	507	$3.0e-7$	$7.5e-8$	$11382 \pm 5$
log10	common logarithm	$\log_{10}(x)$	3	550	$1.7e-7$	$4.8e-8$	$10343 \pm 5$
log2	binary logarithm	$\lg(x)$	3	193	$3.9e-5$	$6.5e-6$	$4402 \pm 5$
ilogb	integer binary log	$\lfloor \lg(x) \rfloor$	0	128	0	0	$3306 \pm 1$
ilog10	integer base-10 log	$\lfloor \log_{10}(x) \rfloor$	0	40	0	0	$1158 \pm 5$
sqrt	square root	$\sqrt{x}$	3	999	$1.2e-4$	$2.6e-5$	$34911 \pm 4$
cbrt	cube root	$\sqrt[3]{x}$	3	765	$1.7e-5$	$2.1e-6$	$24326 \pm 4$
qtrt	quartic root	$\sqrt[4]{x}$	3	379	$1.6e-5$	$6.3e-6$	$9344 \pm 5$
isqrt	reciprocal square root	$1/\sqrt{x}$	3	330	$3.5e-6$	$1.6e-7$	$4174 \pm 3$
icbrt	reciprocal cube root	$1/\sqrt[3]{x}$	3	192	$2.0e-5$	$3.7e-6$	$1082 \pm 4$
iqtrt	reciprocal quartic root	$1/\sqrt[4]{x}$	3	367	$3.5e-6$	$1.6e-7$	$5862 \pm 4$
reciprocal	reciprocal	$1/x$	3	561	$4.6e-5$	$4.6e-6$	$1051 \pm 3$
erf	Gaussian error function	$(2/\sqrt{\pi}) \int_0^x e^{-t^2} dt$	3	70	$3.0e-8$	$1.7e-8$	$626 \pm 2$
erfc	complementary erf	$1 - \text{erf}(x)$	3	92	$4.2e-8$	$1.6e-8$	$772 \pm 2$
abs	absolute value	$ x $	1	2	0	0	$114 \pm 0$
signum	sign number	$\text{sgn}(x)$	0	3	0	0	$114 \pm 0$
positive	test strictly positive	$[x > 0]$	0	2	0	0	$114 \pm 0$
negative	test strictly negative	$[x < 0]$	0	2	0	0	$114 \pm 0$
nonneg	test non-negative	$[x \geq 0]$	0	2	0	0	$114 \pm 0$
nonpos	test non-positive	$[x \leq 0]$	0	2	0	0	$114 \pm 0$
zero	test exactly zero	$[x = 0]$	0	2	0	0	$57 \pm 0$
nonzero	test non-zero	$[x \neq 0]$	0	2	0	0	$57 \pm 0$
clz	count leading zeros	$\text{clz}(x)$	0	49	0	0	$1665 \pm 1$
clrsb	count redundant sign bits	$\text{clrsb}(x)$	0	95	0	0	$3207 \pm 1$
ReLU	rectified linear unit	$\max(0, x)$	1	2	0	0	$114 \pm 0$
ReLU6	clipped ReLU	$\min(\max(0, x), 6)$	1	3	0	0	$128 \pm 2$
LeakyReLU	leaky ReLU	$\max(0, x) + \min(0, x/100)$	1	2	0	0	$114 \pm 0$
ReLU <sup>2</sup>	squared ReLU	$\max(0, x^2)$	2	2	0	0	$114 \pm 0$
GELU	Gaussian error linear unit	$x(1 + \text{erf}(x/\sqrt{2}))/2$	3	81	$6.0e-6$	$2.5e-7$	$712 \pm 2$
HardELISH	"hard" ELISH	$\min(e^x - 1,  x ) \max(0, \min(1, \frac{x+1}{2}))$	3	38	$4.2e-8$	$1.2e-8$	$351 \pm 2$
Hardshrink	"hard" shrink	$[( x  < 1 ? 0 : x)$	1	3	0	0	$182 \pm 1$
Hardsigmoid	"hard" logistic sigmoid	$x < -3 ? 0 : (x > 3 ? 1 : (x+3)/6)$	1	3	0	0	$128 \pm 2$
Hardswish	"hard" swish	$x < -3 ? 0 : (x > 3 ? 1 : x \cdot (x+3)/6)$	2	3	0	0	$128 \pm 1$
Hardtanh	"hard" hyperbolic tangent	$x < -1 ? -1 : (x > 1 ? 1 : x)$	1	3	0	0	$126 \pm 2$
Softplus	"soft" plus	$\ln(1 + e^x)$	3	94	$1.2e-7$	$2.2e-8$	$960 \pm 2$
Softminus	"soft" minus	$x - \text{Softplus}(x)$	3	93	$1.2e-7$	$2.2e-8$	$946 \pm 2$
Softsign	"soft" sign	$x/(1 +  x )$	3	182	$6.2e-7$	$3.9e-8$	$2247 \pm 3$
Softshrink	"soft" shrink	$\text{sgn}(x) \cdot \max( ( x ) - 1, 0)$	1	3	0	0	$125 \pm 1$
ELU	exponential linear unit	$\max(\alpha e^x - 1, x)$	3	46	$3.0e-8$	$1.7e-8$	$496 \pm 3$
sigmoid	logistic sigmoid	$1/(1 + e^{-x})$	3	98	$1.1e-7$	$1.4e-8$	$991 \pm 2$
SiLU	sigmoid linear unit	$x \cdot \text{sigmoid}(x)$	3	108	$1.2e-7$	$2.6e-8$	$1046 \pm 2$
CELU	continuously differentiable ELU	$\max(0, x) + \min(0, e^x - 1)$	3	47	$3.0e-8$	$1.7e-8$	$507 \pm 2$
ELISH	exponential-linear squash-ing	$x < 0 ? \frac{x}{1+e^x} : \frac{e^x-1}{1+e^x}$	3	115	$1.2e-7$	$2.4e-8$	$1093 \pm 3$
Mish	Misra's swish	$x \cdot \tanh(\text{Softplus}(x))$	3	106	$1.2e-7$	$1.8e-8$	$1005 \pm 3$
LeCunTanh	LeCun's hyperbolic tangent	$1.7159 \tanh(\frac{x}{2})$	3	89	$4.2e-8$	$2.3e-8$	$860 \pm 2$
TanhExp	tanh-exponential	$x \cdot \tanh(e^x)$	3	100	$6.0e-8$	$1.1e-8$	$931 \pm 3$
TanhShrink	tanh shrink	$x - \tanh(x)$	3	99	$8.4e-8$	$5.9e-8$	$852 \pm 2$
Serf	log-softplus error	$x \cdot \text{erf}(\text{Softplus}(x))$	3	102	$5.9e-8$	$1.5e-8$	$943 \pm 2$
logsigmoid	natural ogarithm of sigmoid	$\ln \text{sigmoid}(x)$	3	67	$4.1e-5$	$7.2e-6$	$682 \pm 2$
tanh	hyperbolic tangent	$(e^x - e^{-x})/(e^x + e^{-x})$	3	84	$3.0e-8$	$1.7e-8$	$770 \pm 3$

<sup>†</sup>Assuming 64-bit fixed-point arithmetic using 16-bits to represent the fractional part.<sup>‡</sup>This function has one or more poles; error measurements exclude points with a distance less than  $1.5e-3$  from a pole.<sup>‡</sup>This is a periodic function for which only the principle domain is included in the LUT

Table 5.1 summarizes selected gadgets (65 in all) supported out-of-the-box by GROTTO, including efficiency metrics (polynomial degree, number of parts in  $\vec{P}$ , and expected number of half-PRG invocations used by the prefix-parity algorithm) alongside fidelity metrics (maximum error and root-mean-squared approximation error) for each gadget. This table also shows polynomial degrees and lookup table sizes for each gadget when using 64-bit fixed-point arithmetic with 16 fractional bits.

### Performance benchmarks

The remainder of this section reports our findings from a series of experiments we ran on a workstation equipped with 128GiB of RAM and an Intel Core i9-12900KS processor running Ubuntu 22.04. We ran all experiments for 100 trials and report the sample mean alongside the sample standard deviation over those 100 trials. (We express this as *mean*  $\pm$  *stdev*.) All numbers are reported to one significant figure in the sample standard deviation.

We compare GROTTO against the recent work closest to it, namely LLAMA [35]. Table 5.2 presents the communication cost and running times for various gadgets and network conditions. In each experiment, the peers perform 10,000 evaluations in parallel. For each experiment, we measured GROTTO with and without the carry-in correction to demonstrate the cost of this (often optional) step.

The first three gadgets—*reciprocal square root* (isqrt), *hyperbolic tangent* (tanh), and the *logistic sigmoid* function (sigmoid)—are also provided by the reference implementation of LLAMA, albeit only with 16-bit words. In addition to the three above-mentioned functions, we also benchmark GROTTO on the *square root* (sqrt) and *base-10 logarithm* (log10) functions. We note that sqrt is the costliest function among those listed in Table 5.1 regarding expected half-PRG calls.

Our experiments highlight the practical benefits of the simple DPFs of GROTTO over the heavier DCFs used by LLAMA, with the gap between GROTTO and LLAMA’s preprocessing phases growing rapidly as network conditions deteriorate. Indeed, in all instances, GROTTO’s preprocessing time and communication costs are but a fraction of what LLAMA uses; online computation times are consistently lower as well, with even GROTTO on 64-bit words competing against LLAMA on 16-bit words

Table 5.2: Cost comparison for 10,000 runs of selected GROTTO- versus LLAMA-based function evaluations

Function	Scheme	Bits	Parts	Degree	Preprocessing				Online				Rounds
					1000mbit/0ms	100mbit/50ms	10mbit/100ms	bandwidth	1000mbit/0ms	100mbit/50ms	10mbit/100ms	bandwidth	
sigmoid	LLAMA	16=7.9	34	2	17 000 ± 1000ms	31 500 ± 600 ms	281 800 ± 600 ms	157.39MiB	4700 ± 400 ms	5600 ± 200 ms	7300 ± 300 ms	351.57KiB	3
	GROTTO				103 ± 2 ms	500 ± 10 ms	1060 ± 30 ms	2.91MiB	105 ± 5 ms	515 ± 5 ms	1062 ± 6 ms	625 KiB	3
	GROTTO+				145 ± 4 ms	630 ± 20 ms	1410 ± 70 ms	6.17MiB	144 ± 2 ms	640 ± 40 ms	1405 ± 3 ms	703.13KiB	4
	LLAMA	16=5.11	34	2	18 000 ± 1000ms	31 600 ± 700 ms	282 600 ± 500 ms	157.39MiB	4800 ± 400 ms	5700 ± 200 ms	8800 ± 300 ms	351.57KiB	3
	GROTTO				110 ± 10 ms	508 ± 9 ms	1070 ± 40 ms	2.91MiB	112 ± 6 ms	515 ± 4 ms	1063 ± 8 ms	625 KiB	3
	GROTTO+				148 ± 7 ms	635 ± 9 ms	1410 ± 60 ms	6.17MiB	148 ± 7 ms	640 ± 30 ms	1406 ± 6 ms	703.13KiB	4
	LLAMA	16=3.13	15	2	6900 ± 400 ms	16 800 ± 900 ms	136 100 ± 600 ms	157.39MiB	3600 ± 300 ms	4200 ± 100 ms	5200 ± 300 ms	351.57KiB	3
	GROTTO				129 ± 3 ms	500 ± 10 ms	1070 ± 10 ms	2.91MiB	103 ± 4 ms	511 ± 2 ms	1063 ± 7 ms	625 KiB	3
	GROTTO+				139 ± 6 ms	640 ± 10 ms	1410 ± 20 ms	6.86MiB	144 ± 8 ms	640 ± 30 ms	1390 ± 30 ms	703.13KiB	4
	LLAMA	64=48.16	98	3	—	—	—	—	—	—	—	—	—
	GROTTO				179 ± 6 ms	960 ± 60 ms	7447 ± 3 ms	13.73MiB	720 ± 80 ms	3500 ± 300 ms	7400 ± 700 ms	1875 KiB	3
	GROTTO+				337 ± 7 ms	1870 ± 70 ms	15 200 ± 700 ms	28.99MiB	920 ± 30 ms	5100 ± 600 ms	9500 ± 500 ms	1953.13KiB	4
invsqrt	LLAMA	16=6.10	10	2	750 ± 60 ms	742 ± 40 ms	750 ± 50 ms	110.40MiB	900 ± 60 ms	1500 ± 100 ms	2360 ± 20 ms	351.57KiB	3
	GROTTO				102 ± 1 ms	610 ± 10 ms	1080 ± 70 ms	2.91MiB	103 ± 5 ms	520 ± 20 ms	1064 ± 6 ms	625 KiB	3
	GROTTO+				148 ± 2 ms	620 ± 7 ms	1400 ± 50 ms	6.83MiB	141 ± 8 ms	620 ± 20 ms	1400 ± 20 ms	703.13KiB	4
	LLAMA	16=4.12	10	2	5900 ± 300 ms	11 800 ± 300 ms	95 500 ± 600 ms	111.08MiB	1150 ± 70 ms	1710 ± 60 ms	7800 ± 300 ms	351.57KiB	3
	GROTTO				105 ± 3 ms	625 ± 8 ms	1060 ± 20 ms	2.91MiB	102 ± 3 ms	510 ± 30 ms	1061 ± 8 ms	625 KiB	3
	GROTTO+				141 ± 2 ms	620 ± 20 ms	1400 ± 100 ms	7.20MiB	142 ± 9 ms	620 ± 30 ms	1400 ± 20 ms	703.13KiB	4
	LLAMA	64=48.16	330	3	—	—	—	—	—	—	—	—	—
	GROTTO				178 ± 6 ms	940 ± 50 ms	7448 ± 1 ms	13.73MiB	980 ± 60 ms	3500 ± 500 ms	7800 ± 600 ms	1875 KiB	3
	GROTTO+				338 ± 8 ms	1880 ± 70 ms	15 000 ± 900 ms	28.99MiB	1200 ± 90 ms	4100 ± 60 ms	9900 ± 700 ms	1953.13KiB	4
tanh	LLAMA	16=7.9	12	2	6500 ± 300 ms	13 500 ± 500 ms	111 800 ± 600 ms	129.13MiB	1260 ± 80 ms	1830 ± 70 ms	8300 ± 300 ms	351.57KiB	3
	GROTTO				100 ± 3 ms	500 ± 10 ms	1060 ± 20 ms	2.91MiB	103 ± 4 ms	512 ± 5 ms	1064 ± 6 ms	625 KiB	3
	GROTTO+				150 ± 10 ms	630 ± 50 ms	1410 ± 60 ms	6.17MiB	139 ± 6 ms	610 ± 40 ms	1404 ± 4 ms	703.13KiB	4
	LLAMA	16=5.11	20	2	10 500 ± 600 ms	21 000 ± 1100ms	174 300 ± 600 ms	200.64MiB	2500 ± 200 ms	3300 ± 0 ms	9000 ± 400 ms	351.57KiB	3
	GROTTO				108 ± 7 ms	505 ± 5 ms	1000 ± 90 ms	2.91MiB	103 ± 5 ms	512 ± 3 ms	1060 ± 8 ms	625 KiB	3
	GROTTO+				150 ± 1 ms	640 ± 10 ms	1410 ± 90 ms	6.86MiB	143 ± 8 ms	630 ± 20 ms	1390 ± 30 ms	703.13KiB	4
	LLAMA	64=48.16	84	3	—	—	—	—	—	—	—	—	—
	GROTTO				178 ± 6 ms	950 ± 60 ms	7447 ± 2 ms	13.73MiB	710 ± 80 ms	3500 ± 500 ms	7200 ± 700 ms	1875 KiB	3
	GROTTO+				337 ± 7 ms	1860 ± 70 ms	14 900 ± 900 ms	28.99MiB	840 ± 30 ms	4600 ± 500 ms	9500 ± 1100 ms	1953.13KiB	4
log10	GROTTO	64=48.16	84	3	178 ± 6 ms	950 ± 60 ms	7447 ± 2 ms	13.73MiB	2000 ± 100 ms	3600 ± 400 ms	7300 ± 900 ms	1875 KiB	3
	GROTTO+				2200 ± 200 ms	4500 ± 500 ms	14 900 ± 900 ms	28.99MiB	2200 ± 200 ms	4500 ± 500 ms	8800 ± 600 ms	1953.13KiB	4
sqrt	GROTTO	64=48.16	84	3	176 ± 5 ms	940 ± 60 ms	7446 ± 2 ms	13.73MiB	5700 ± 500 ms	6400 ± 400 ms	8090 ± 60 ms	1875 KiB	3
	GROTTO+				337 ± 7 ms	1860 ± 70 ms	14 900 ± 900 ms	28.99MiB	6100 ± 600 ms	7900 ± 700 ms	10 500 ± 1000 ms	1953.13KiB	4

for all of the common gadgets.

## Analysis

The experimental results summarized in Table 5.2 demonstrate that DPFs with the parity-segment approach can handily outperform DCFs in terms of both communication and computation costs.

In particular, consider the approximation of  $f: \mathbb{R} \rightarrow \mathbb{R}$  via a piece-wise polynomial function comprising  $P$  parts and polynomials of degree  $d$ . Whereas the DCFs that LLAMA uses to map inputs to parts each occupy about

$$\lambda \cdot (\lg N + 1) + P \cdot (\lg N)^2 \cdot (d + 1) + 2P \cdot (d + 1) \lg N$$

$$\in O(P \cdot \lg N^2 \cdot d),$$

the corresponding DPF only uses about  $\lambda \cdot (\lg N - \lg \lambda) \in O(\lambda \cdot \lg N)$  bits. For computation, the required number of half-PRG evaluations shrinks from  $O(P \cdot \lg N)$  to  $o(P \cdot \lg N)$  alongside a corresponding shrinking of the hidden constants.

More generally, one can view DPFs with the parity-segment segment technique *à la* GROTTO as being functionally equivalent to—albeit much more compact and computationally efficient than—the DCFs employed by LLAMA, save for two subtle differences:

1. First, on the positive side, whereas the entire LUT (i.e., both the  $x$ - and  $y$ -coordinates of each “point”) are hardcoded by the DCF dealer at generation time, DPFs with GROTTO’s parity-segment approach allows dynamically choosing those points at evaluation time. This results in a functionality-independent preprocessing step that should be much easier to emulate in 2-party settings.
2. Second, on the negative side, using DPFs with 1-bit outputs necessitates an extra round of interaction (for sign correction) relative to a DCF. For polynomials of degree  $d \geq 1$ , GROTTO can “hide” this round by piggybacking on the rounded need for polynomial evaluation; however, this extra round appears unavoidable when the  $y$ -coordinates are scalars.

As a general rule of thumb, replacing a DCF with a DPF results in a net performance gain if (and essentially only if) the additional communication round is not a bottleneck, such as when it can happen in parallel with some other, preexisting message flow.

In terms of round complexity and online communication, GROTTO is also competitive with LLAMA but cannot compete with DCFs’ non-interactive evaluation. In particular, GROTTO incurs the same round complexity as LLAMA, but has a noticeably higher online communication cost, owing to our use of cubic (rather than quadratic) polynomials and our strategy of lifting shares to a larger ring. The upshot of this higher online communication is the capacity for faster and more accurate approximations: GROTTO’s approximations for all three functions exhibit a maximum error less than the fixed-point numbers can represent, whereas LLAMA tolerates deviations of up to 4 units in the last place (ULPs) of error.

We stress that all of the times we present explicitly ignore the network communication time. This is done for consistency with the available implementation of LLAMA and to remove the communication overhead that dominates the running time of both LLAMA and GROTTO alike. Besides, typical Internet latency is easily four to five orders of magnitude higher than GROTTO’s running time, making the overhead of GROTTO impossible to separate from the variance in the network latency.

## 5.5 Conclusion

In this chapter, we introduced GROTTO, a framework and C++ library for space- and time-efficient  $(2 + 1)$ -party piecewise polynomial evaluation on secrets additively shared over  $\mathbb{Z}_{2^{n+k}}$ . GROTTO improves on the state-of-the-art approaches based on DCFs in almost every metric, offering asymptotically superior communication and computation costs with comparable round complexity. At the heart of GROTTO is a novel observation about the structure of the most compact DPFs from the literature and our prefix-parity algorithm, which leverages this structure to do with a single DPF what others require DCFs to do.

## Chapter 6

# Beyond GROTTO: DPFs for bit decomposition in MPC

As described in [Chapter 5](#), GROTTO enables fast computation of non-linear functionalities in MPC. However, this only scratches the surface of what GROTTO can be used for. This chapter shows how GROTTO can be used to simultaneously evaluate multiple functionalities in parallel on a single input at a much lower cost than employing separate GROTTO instances in parallel for each functionality. We then proceed to show how this parallel composition technique can be applied to produce a highly optimized bit decomposition algorithm which dramatically outperforms the techniques for arithmetic to XOR share conversions used in all existing systems [22, 50, 54]. A natural application of this protocol is a new ABY-like framework which integrates DPFs for share conversion and function computation. This approach gains a significant performance advantage over ABY framework variants through its superior bit decomposition protocol and by using GROTTO to evaluate non-linear functionalities.

### 6.1 Function composition

Recall the basic flow of GROTTO's segment-parity-finding algorithm on an ordered sequence  $E$  of endpoints partitioning the bitstring  $S$  into  $P$  segments (up to cyclic rotation) and a shift  $s$ :

1. invoke the prefix-parity algorithm to compute the parity  $p_i$  of every prefix ending at an endpoint  $e_i + s$  with  $e_i$  from  $E$ , and then
2. for each successive pair  $(p_i, p_{i+1})$  of prefix parities, output the segment parity

$$\text{parity}(S[e_i \dots e_{i+1}]) = \begin{cases} p_i \oplus p_{i+1} & \text{if } p_i < p_{i+1}, \text{ and} \\ p_i \oplus p_{i+1} \oplus 1 & \text{otherwise.} \end{cases} \quad (6.1)$$

The “otherwise” clause in [Equation \(6.1\)](#) handles the case where a segment wraps cyclically, which can happen for at most one of the  $P$  segments induced by  $E$ .

Our first observation concerns the simultaneous evaluation of several functions at the same  $x$ . It follows from the fact that the above-sketched prefix-parity-finding procedure merely takes arbitrary (sorted) endpoint sequences as input; the DPFs are in no way tailored toward implementing any specific function, apart from having the appropriate depth.

**Observation 7.** *If  $f_1, \dots, f_c$  is a collection of  $c$  univariate functions all sharing a common domain and  $[x]$  is an additively shared input, then we can securely evaluate the sequence of secret-shared images  $[f_1(x)], \dots, [f_c(x)]$  using a single DPF.*

We refer to such simultaneous evaluations using a single DPF as a *composition* of the functions  $f_1, \dots, f_k$ .

**Parallel versus sequential composition** The most obvious way to compose  $k$  functions in GROTTO is to individually shift the segment endpoints for each function and then to run the entire segment-parity-finding algorithm  $k$  times in a row (i.e., once for each function). Such a *sequential composition* of functions maximally amortizes the (both communication and computation) cost associated with DPF generation and distribution—but not the cost of segment-parity finding.<sup>1</sup>

Specifically, recall that the prefix-parity algorithm incurs an amortized cost per prefix that decreases as the number of prefix parities to compute increases, with the rate of this decrease depending on the distribution of prefix lengths. Roughly speaking, segment endpoints that are more densely packed imply greater amortization as

<sup>1</sup>We stress that the prefix-parity step is non-interactive, so our use of “sequential” here does *not* imply sequential message passing between shareholders.



is shown in [Appendix A](#). Indeed, in the extreme case where two or more consecutive prefixes terminate within a common leaf node, only the lexicographically first among them incurs *any* non-trivial computation cost. This suggests that a superior approach to sequential composition is *parallel composition*, wherein the prefix-parity algorithm is run just once on a merged (and deduplicated) sequence comprising the shifted endpoints for all of the  $k$  functions.

[Figure 6.1](#) illustrates the two styles of composition for a pair of functions (namely, the principle domains of  $\sin(x)$  and  $\cos(x)$ ). In this example, both functions are approximated by simple piecewise-linear functions respectively comprising 13 and 14 parts (or, equivalently, 14 and 15 endpoints, indicated in [Figure 6.1a](#) by the markers along each curve).

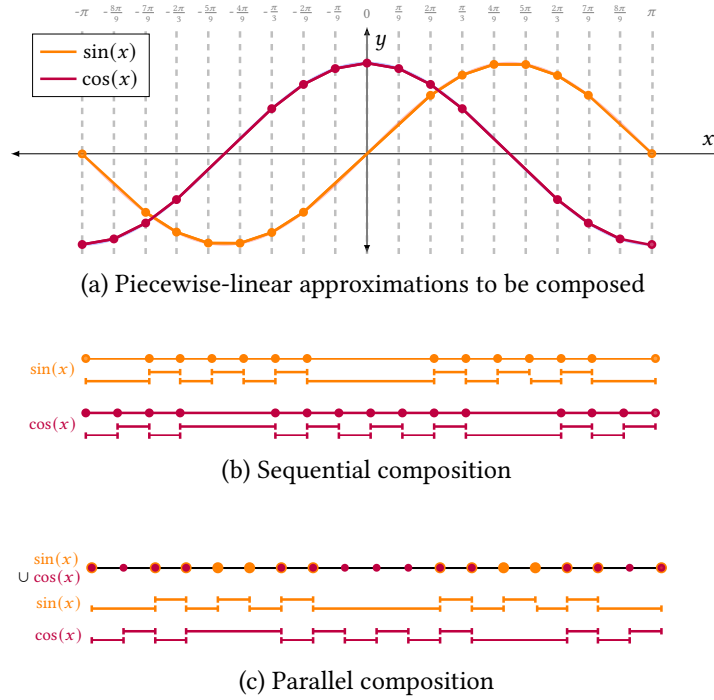


Figure 6.1: Sequential versus parallel composition of piecewise-linear approximations to  $\cos(x)$  and  $\sin(x)$  on their principle domain (namely  $[-\pi \dots \pi]$ ).

The *sequential* composition depicted in [Figure 6.1b](#) invokes the prefix-parity algorithm twice, on endpoint sequences of length 14 and 15, before computing 13

and 14 one-bit XORs to compute the desired segment parities. Using [Theorem 3](#), we can compute a worst-case bound for the cost of this approach as

$$(13n - \sum_{i=2}^{13} \lfloor \lg(i-1) \rfloor) + (14n - \sum_{i=2}^{14} \lfloor \lg(i-1) \rfloor) = 27n - 53$$

half-PRG evaluations, plus a handful of XORs. Meanwhile, the *parallel* composition depicted in [Figure 6.1c](#) invokes the prefix-parity algorithm just once on an endpoint sequence of length 19, before computing 13 and 14 one-bit XORs analogous to those in the sequential composition;<sup>2</sup> thus, the worst-case bound for the cost of parallel composition is just

$$(19n - \sum_{i=2}^{19} \lfloor \lg(i-1) \rfloor) = 19n - 46$$

half-PRG evaluations, plus a handful of XORs. For  $n = 57$  (i.e., 64-bit shares and 128-bit leaves), this works out to a reduction of the worst-case bound by  $1 - \frac{19 \cdot 57 - 46}{27 \cdot 57 - 53} \approx 30.2\%$  half-PRG evaluations relative to sequential composition.

## 6.2 Bit extraction

0 =	0	0	0
1 =	0	0	1
2 =	0	1	0
3 =	0	1	1
4 =	1	0	0
5 =	1	0	1
6 =	1	1	0
7 =	1	1	1
	bit <sub>1</sub>	bit <sub>2</sub>	bit <sub>3</sub>

Figure 6.2: Bucket layout for bit extraction in GROTTO

Before trying to perform bit decomposition, consider the problem of extracting XOR shares of the MSB of an  $n$ -bit arithmetic secret shared value. Recall that the first  $2^{n-1}$  unsigned  $n$ -bit numbers will have the MSB cleared and the other  $2^{n-1}$  such values will have the MSB set. This can be easily represented exactly as a GROTTO functionality in which the spline for the functionality is simply the constant 0 for

<sup>2</sup>In fact, the two approximations have four segments in common, so that only  $13 + 14 - 4 = 23$  such one-bit XORs are strictly required.

all inputs less than  $2^{n-1}$  and the constant 1 for all inputs greater than or equal to  $2^{n-1}$ . This same concept easily extends to the  $i$ th most significant bit (where the MSB is the case  $i = 1$ ) which will divide the range into  $2^i$  equal size pieces with values alternating between 0 and 1 as is shown in [Figure 6.2](#). We will denote the function which extracts the  $i$ th MSB of  $x$  as  $\text{bit}_i(x)$ .

While this technique will correctly extract any bit in the input number, the number of buckets in the basic bit extraction GROTTO functionality grows exponentially as the bit being extracted is further away from the MSB. In order to improve the efficiency of bit extraction, we can reduce the number of buckets in the functionality while simultaneously using smaller DPFs by removing excess leading bits from the input number. To show that it is possible to remove leading bits from a secret shared value using a completely offline computation, consider  $n$ -bit secret shares of  $[x]$ , we know that  $x \equiv [x]_0 + [x]_1 \pmod{2^n}$ . Which means that  $x + q \cdot 2^n = [x]_0 + [x]_1$  for some  $q \in \mathbb{Z}$ . Then, for any non-negative integer  $m < n$ , we have  $x + (q \cdot 2^{n-m}) \cdot 2^m = [x]_0 + [x]_1$ , so  $x \equiv [x]_0 + [x]_1 \pmod{2^m}$ . As a result, reducing the shares  $[x]_0$  and  $[x]_1$  to be  $m$ -bit values produces correct  $m$ -bit shares of  $x$  modulo  $2^m$ . This means that the secret shares representing the  $n$ -bit input value to a GROTTO function can be reduced to  $m$ -bit shares for any  $0 < m \leq n$ . The  $m$ -bit shares can then be used with an  $m$ -bit DPF to evaluate GROTTO functions exactly as described in [Chapter 5](#). In the case of extracting the  $i$ th MSB of  $x$ , the input can be reduced to have  $n - i + 1$  bits. With the DPF only representing the least significant  $n - i + 1$  bits, the  $i$ th MSB of the original input  $x$  becomes the most significant bit of the DPF's distinguished point,  $x \pmod{2^{n-i+1}}$ . Furthermore, when we remove the most significant  $b$  bits from the input and use only the low order  $n - b$  bits for the DPF, this reduces the number of segments for  $\text{bit}_i(x)$  from  $2^i$  to  $2^{i-b}$ .

### 6.3 Bit decomposition

In order to perform a bit decomposition, we want to compose multiple bit extraction functionalities together to retrieve all bits of the input number. More generally, given a set of bits  $B \subseteq \{1, 2, \dots, n\}$  such that we want to compute  $\text{bit}_b(x)$  for all  $b \in B$ , we can reduce the DPF size to  $n - c + 1$ -bits long where  $c = \min(B)$ . This means that the most significant bit to be extracted will always be the MSB of the reduced bit

length input. This reduces the number of segments used to extract bit  $i \in B$  from  $2^i$  to  $2^{i-c+1}$  as explained in [Section 6.2](#). Now, we want to determine the number of prefix parities that we need to compute when extracting all bits listed in  $B$ .

**Theorem 7.** *Given  $\text{bit}_i$  and  $\text{bit}_j$  where  $i < j$ , the sets of endpoints for these functionalities on an  $n$ -bit DPF are  $E_i$  and  $E_j$  respectively where  $E_i \subseteq E_j$ .*

*Proof.* To extract  $\text{bit}_i$  there are  $2^i$  segments equally distributed to cover the entire domain, of size  $2^n$ . Therefore, the segments used are of length  $2^{n-i}$ . This means that  $E_i = \{a \cdot 2^{n-i} \mid 0 \leq a \leq 2^i\}$ . By an analogous argument,  $E_{i+1} = \{a \cdot 2^{n-i-1} \mid 0 \leq a \leq 2^{i+1}\}$ . Suppose,  $b = a \cdot 2^{n-i} \in E_i$ , so  $0 \leq a \leq 2^i$ . Then,  $b = (2a) \cdot 2^{n-i-1}$  and  $0 \leq 2a \leq 2 \cdot 2^i = 2^{i+1}$ . Therefore,  $b \in E_{i+1}$ .

By repeated application of this fact, we have that  $E_i \subseteq E_{i+1} \subseteq \dots \subseteq E_j$ .  $\square$

**Corollary 3.** *Given a set of bits  $B$  with  $b = \max(B)$ , the set of endpoints for  $\text{bit}_b$ , denoted  $E_b$ , is the set of all endpoints for all GROTTO functions  $\text{bit}_i$  where  $i \in B$ .*

In [Corollary 3](#), we see that the number of prefix endpoints, and thus the computation cost, for extracting multiple bits from a single DPF of a given size is entirely dependent on the number of endpoints being used for the least significant bit that is being extracted. Now, if a single DPF is used to extract all bits from an  $n$ -bit value, then the number of endpoints will be the number of endpoints required to extract the least significant bit,  $\text{bit}_n(x)$ . Thus, the total number of segments will be  $2^n$  which is infeasible for common bit lengths such as  $n = 64$ .

To avoid the high cost of extracting all bits from a single DPF, we can use multiple DPFs to extract different bits. Since the computational cost of prefix-parity does not depend on the number of bits being extracted between the most and the least significant bits that are being extracted, each DPF will be used to extract a contiguous range of bits going from a specified most significant bit to a specified least significant bit. Each DPF then has its input size set such that the most significant bit to be extracted from that DPF is the most significant bit of the distinguished point. Therefore, each range is extracted with as few prefix parity computations as possible. This reduces the overall number of segments required for the prefix-parity computation at the cost of using multiple DPFs. To tune the cost of the bit decomposition, the number of DPFs can be increased or decreased based on the

importance of computational cost versus the number of precomputed DPFs in the specified scenario.

Formally, in order to perform bit decomposition on an  $n$ -bit additive secret shared number  $[x]$ , we choose the number of DPFs  $\gamma$  and  $\gamma$  dividing points  $1 = m_1 < m_2 < \dots < m_\gamma = n$ . We will then generate  $\gamma$  DPFs for the bit extraction, denoted  $d_1, d_2, \dots, d_\gamma$ . DPF  $d_i$  will be used to extract the  $m_i$  to  $m_{i+1} - 1$  most-significant bits. Since the most significant bit being extracted is the  $m_i$  most significant bit, the input can be reduced to  $n - m_i + 1$  bits. Using the most recent version of the Boyle–Gilboa–Ishai DPF construction, this results in a DPF key of size  $(n - m_i + 1) \cdot (\lambda + 2) + 1$  bits [14]. Now, the least significant bit being extracted is the  $m_{i+1} - 1$  most-significant bit of the original  $x$ , which is the  $m_{i+1} - m_i$  most-significant bit of the  $n - m_i + 1$ -bit  $[x]$ . Therefore, the resulting bit extraction functionality will have  $2^{m_{i+1} - m_i}$  parities to compute.

The use of multiple DPFs of different bit lengths for bit decomposition leads to the observation that each bit decomposition DPF can be chosen with the same random distinguished point  $\alpha$  truncated to the number of bits in the DPF. As a result, the shift  $x - \alpha$  computed over  $n$ -bit values can be used to shift all of the DPFs to get the correct values. This removes the need for separate shifts to be computed for every DPF being used. As a result, using multiple DPFs will increase the precomputation required in the protocol, but it will not change the online communication cost of the bit decomposition protocol, as it would if each DPF needed a different shift.

## 6.4 Performance evaluation

As stated in the previous section, in order to decompose an  $n$ -bit additive secret sharing using  $\gamma$  DPFs, the DPF for bits  $m_i$  to  $m_{i+1} - 1$  will have a key of size  $(n - m_i + 1) \cdot (\lambda + 2) + 1 = (n + 1)\lambda + 2n + 3 - (\lambda + 2)m_i$  bits. The total size of the DPFs is then  $\sum_1^\gamma ((n + 1)\lambda + 2n + 3 - (\lambda + 2)m_i) = \gamma((n + 1)\lambda + 2n + 3) - (\lambda + 2) \sum_1^\gamma m_i$  bits. Generally, the endpoints will be equally distributed, with  $m_i = i \cdot \frac{n}{\gamma}$ . Thus, the total

size (in bits) of the precomputed DPFs for each party becomes

$$\begin{aligned}
 \gamma((n+1)\lambda + 2n + 3) - (\lambda + 2) \sum_1^\gamma i \cdot \frac{n}{\gamma} &= \gamma((n+1)\lambda + 2n + 3) - (\lambda + 2) \frac{n}{\gamma} \sum_1^\gamma i \\
 &= \gamma((n+1)\lambda + 2n + 3) - (\lambda + 2) \cdot \frac{n}{\gamma} \cdot \frac{\gamma(\gamma + 1)}{2} \\
 &= \gamma((n+1)\lambda + 2n + 3) - (\lambda + 2) \cdot \frac{n(\gamma + 1)}{2} \\
 &= \gamma \left( \frac{\lambda n}{2} + \lambda + n + 3 \right) - n \left( \frac{\lambda}{2} + 1 \right)
 \end{aligned}$$

In addition to the precomputed DPFs, each party receives an  $n$ -bit share of the distinguished point  $[\alpha]$ . Only one such distinguished point share is required, as specified earlier, because each DPF uses the same random distinguished point reduced to the appropriate bitlength. Therefore, the total number of precomputed bits received by each party is  $n + \gamma \left( \frac{\lambda n}{2} + \lambda + n + 3 \right) - n \left( \frac{\lambda}{2} + 1 \right)$ .

During the online phase, only one round of communication is required. This round computes the shift value  $x - \alpha$ . No polynomial evaluation or sign correction is required because the GROTTO functionality uses constant outputs rather than polynomials and those outputs are XOR shares. As previously discussed, only one shift needs to be computed. Thus, the only online cost is having party  $i$  send  $n$ -bit share  $[x - \alpha]_i$ .

System	Precomputation	Communication	Rounds
ABY	$4n\lambda$	$2n\lambda + n$	2
ABY2.0	$4n\lambda + n$	$n\lambda + n$	2
ABY <sup>3</sup>	0	$n + n \log n$	$1 + \log n$
GROTTO	$n + \gamma \left( \frac{\lambda n}{2} + \lambda + n + 3 \right) - n \left( \frac{\lambda}{2} + 1 \right)$	$n$	1

Table 6.1: Arithmetic to Boolean Secret Share Conversion Performance Comparison

Table 6.1 shows how GROTTO-based bit decomposition has a lower communication cost and round complexity than existing ABY-style frameworks. All solutions prior to GROTTO work by XOR secret sharing the individual arithmetic shares  $[x]_i$  of the secret value  $x$  being converted. These shares can then be used to obviously evaluate an addition circuit which produces XOR shares  $([x])_i$  of the original secret shared value. In order to optimize the round complexity, ABY and ABY2.0 perform

this addition circuit evaluation as a garbled circuit to produce Yao shares of the secret value. These Yao shares can then be trivially converted into the corresponding Boolean shares. In ABY<sup>3</sup>, the conversion is performed directly to XOR shares. As a result, the high computation cost of using garbled circuits is avoided, but the resulting round complexity is no longer constant. Instead, the number of rounds is logarithmic in input bitlength  $n$ , because ABY<sup>3</sup> uses a circuit depth optimized parallel prefix adder circuit rather than the standard linear depth adder circuit [50].

## 6.5 Conclusion

This chapter demonstrates how GROTTO can be used to implement a  $(2 + 1)$ -party MPC protocol for performing bit decomposition in an ABY-style framework. By using DPFs rather than standard MPC techniques based upon Boolean circuit evaluation, this technique gains a significant performance advantage over prior works in this area. While this technique, as a  $(2+1)$ -party protocol, relies upon having a third party to provide the precomputed DPFs and shares of the distinguished point  $\alpha$ , the entire protocol can be converted into a standard 2-party protocol by performing the DPF generation as a 2-party protocol. Such a protocol is outlined in [Section 7.4](#).

## Chapter 7

# MPC for DPF verification and generation

In this chapter, we introduce a novel technique for verifying the correctness of DPF keys in MPC protocols. We then show how this protocol can be converted into a Zero knowledge proof of knowledge (ZKPoK) using the MPC-in-the-head paradigm and into an MPC protocol for generating DPF keys. DPF key verification is a critical tool for enabling secure *sender-anonymous messaging* (SAM) protocols built around DPF PIR-writing. When combined with additional optimizations, these MPC and ZKPoK techniques produce the Sabre family of sender-anonymous messaging protocols, which outperform existing sender anonymous bulletin board and mailbox model protocols.

### 7.1 Sender-anonymous messaging preliminaries

A key application described in this chapter is the Sabre family of SAM protocols. SAM protocols are a valuable tool being used to counter the rampant online violation of privacy. State of the art works in this area include Riposte [21] and Express [25]. Such protocols can form the backend for, among other things, whistleblower drop boxes, anonymous email systems, or *Twitter*-like broadcast media with strong and provable privacy guarantees.

Prior work in this space has considered one of two primary settings for SAM



protocols: (i) the *sender-anonymous bulletin board* model and (ii) the *sender-anonymous mailbox* model. Protocols that operate in this first model—exemplified by Riposte [21]—support *Twitter*-like broadcast messaging while severing the link between authors and their messages. Protocols that operate in the second model—exemplified by Express [25]—support *Secure Drop*-like mailboxes into which whistleblowers can leak documents and tips anonymously to journalists, law enforcement, and watchdogs.

### 7.1.1 PIR-writing for SAM protocols

In [Section 2.8.3](#) we introduced the concept of PIR writing using DPFs. This technique is used for writing messages to sender anonymous messaging systems. In particular, Riposte uses a secret shared database to store the messages submitted to it within a given epoch [21]. A similar technique is used in Express to write messages to sender anonymous mailboxes [25]. In both cases, the writer does not perform a read to check if any prior messages have been written to the mailbox or location where it is writing. Instead, the large domain of inputs to the DPF is used to make it statistically infeasible for an attacker to corrupt a given message or mailbox by performing a write targeting to the same location, unless the attacker is told the user’s private index for the message or mailbox. Riposte is able to reduce the domain size by including error correction information in messages being written to the system. Despite the protection that the large domain size offers from targeted message corruption, the basic PIR writing technique presented by Gilboa and Ishai does not provide a mechanism to ensure the correctness of the DPFs received by the servers. As a result, an attacker can send a malformed DPF key to the servers which will result in the entire database being corrupted when the malformed DPF’s output is integrated into the database. To prevent this, systems using DPFs for private writing employ an additional DPF key correctness verification protocol when the writer may be malicious.

### 7.1.2 DPF key correctness verification

In Riposte, two different methods for DPF verification can be employed. The first method is a lightweight three party protocol based on *probabilistic batch testing*

performed by the two database servers and an additional auditing server. This method is designed for a specific DPF construction with  $O(\sqrt{N})$  size keys, rather than the  $O(\log(N))$  size Boyle–Gilboa–Ishai DPFs. In this DPF construction, a key pair is values  $k_A = (b_A, s_A, v)$  and  $k_B = (b_B, s_B, v)$  where  $b_A$  and  $b_B$  differ at exactly one index and  $s_A$  differs from  $s_B$  at the same index. The vector  $v$  is derived from the desired output value and the results produced by applying a pseudorandom generator to the differing values of  $s_A$  and  $s_B$ . The verification works by applying a simple MPC protocol to check whether two pairs of specially constructed test vectors each differ at exactly one index. This verifies the correctness of the DPF key.

Riposte also proposes an alternate verification using  $s$ -party DPFs with  $s$  servers. This DPF construction also produces  $O(\sqrt{N})$  size DPF keys, but it requires the use of a seed-homomorphic pseudorandom generator based upon public-key cryptographic assumptions, which significantly increases the computational cost. A relatively expensive non-interactive zero-knowledge proof of knowledge is then used to verify that the keys are well formed. Much like the two-server DPF verification, this technique is not applicable to the smaller and more efficient Boyle–Gilboa–Ishai construction DPFs.

Express detects malformed messages using a secret-shared non-interactive proof (SNIP) for a linear sketching DPF verification protocol. This linear sketching protocol was first proposed by Boyle et al. as a method for semi-honest servers to verify the correctness of DPF keys which they received [14]. The use of a SNIP makes the verification secure against malicious servers. This method is compatible with the logarithmic size Boyle–Gilboa–Ishai construction DPFs, but it has computation cost  $O(n\lambda)$  where  $n$  is the number of registered mailboxes and  $\lambda$  is the security parameter specifying the number of bits in a mailbox address.

### 7.1.3 Sender-anonymous bulletin boards

In the bulletin board model, two or more semi-honest and non-colluding servers jointly host a shared, sparse database (the “bulletin board”) comprising a large number of write locations called *buckets*. Senders can write messages into arbitrary buckets without disclosing to the servers (or the subsequent readers) which particular messages they penned and which were penned by others. For this to work, senders

must always write their messages to *uniform random* buckets—potentially clobbering messages previously written to those buckets by other senders. Therefore, the number of buckets must be large enough that the probability of accidental collisions is low, implying that the bulletin board requires  $\Omega(m^2)$  buckets to accommodate  $m$  written messages *sans* any expected collisions. Readers obtain messages of interest from the bulletin board either “in the clear” (an undeniably leaky proposition) or via some oblivious means such as PIR [19].

#### 7.1.4 Sender-anonymous mailboxes

In the mailbox model, two or more semi-honest and non-colluding servers jointly host a collection of registered “mailboxes”. Senders can deposit documents into specific mailboxes if and only if they know the corresponding *mailbox addresses*—cryptographically long, unpredictable bit strings—without disclosing to the servers or mailbox owners which particular documents they deposit into which particular mailboxes. The use of registered mailboxes obviates the need both for PIR and for collision-avoidance and -recovery strategies of the sort required in the bulletin board model, potentially enabling constructions that are significantly more performant and scalable. However, it comes at the cost of limiting the use cases to single-recipient applications like email and secure drop boxes.

#### 7.1.5 Relationship with onion routing and mix networks

SAM protocols solve a problem that is related to, yet distinct from those solved by onion routers and mix networks. Onion routing systems like Tor [23] support session-based, bidirectional anonymous links to facilitate low-latency interactive communications such as instant messaging or web browsing; mix networks like MCMix [2] likewise support bidirectional anonymity, but target communications that can tolerate relatively high latency such as email. In both cases, the emphasis is on mitigating the threat of traffic analysis. By contrast, SAM protocols seek only to hide the association between senders and the messages they send—a much less ambitious goal. The upshot of this narrow focus is that it enables SAM protocols to provide very concrete anonymity guarantees (in contrast to the comparatively

“fragile” guarantees of onion routing) while imposing only modest latency (in contrast to the high latency imposed by mix networks).

## 7.2 MPC DPF key verification

The *audit protocol* verifies the well-formedness of DPF keys submitted by writers in both the bulletin board and mailbox model variants of Sabre. It is instantiable in three distinct ways: as (i) a secure  $(2 + 1)$ -party computation, (ii) a 3-verifier MPC-in-the-head SNIP, or (iii) a 2-verifier MPC-in-the-head SNIP. The latter two instantiations build on the first to provide progressively stronger security guarantees. Specifically, (i) by recasting the  $(2 + 1)$ -party computation as a 3-verifier MPC-in-the-head SNIP, the second instantiation maintains the efficiency of the first while removing the threat of a malicious server deviating from the MPC in a bid to violate sender-anonymity, and (ii) by eliminating one of the verifiers, the 2-verifier SNIP makes sender anonymity contingent on a strictly weaker non-collusion assumption (at the expense of some nontrivial communication and computation overhead). We stress that—similar to Riposte and Express—all three instantiations still require semi-honest (and non-colluding) servers to guarantee protocol correctness.

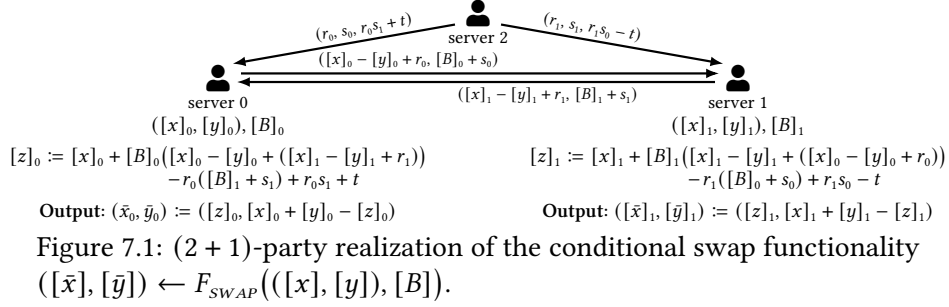
### 7.2.1 Secure $(2 + 1)$ -party auditing

Following [Corollary 1](#) (in [Section 2.7.1](#)), the  $(2 + 1)$ -party audit protocol demonstrates the existence of a 1-path from the root of the binary-tree representation of a DPF to the leaf corresponding to its distinguished input. It does so via an alternating sequence of two simple functionalities: (i) oblivious length-doubling PRG evaluation and (ii) conditional swapping on the components of an ordered pair.

#### Oblivious length-doubling PRG

The oblivious PRG evaluation functionality  $([x], [y]) \leftarrow F_{PRG}([z])$  uses the Matyas–Meyer–Oseas (MMO) one-way compression function [\[49\]](#): Given a block cipher  $\text{Enc}$  with  $\lambda$ -bit blocks, define  $G_\lambda^{2\times} : \{0, 1\}^\lambda \rightarrow \{0, 1\}^\lambda \times \{0, 1\}^\lambda$  via

$$G_\lambda^{2\times}(z) := (\text{Enc}_K(z) \oplus z, \text{Enc}_K(z \oplus 1) \oplus (z \oplus 1)), \quad (7.1)$$



where  $K$  is a fixed (publicly known) key. Sabre uses fixed-key LowMC as the block cipher in Equation (7.1). We remark that using a fixed, publicly known key in LowMC eliminates all non-linear operations outside of the S-boxes, and it provides new opportunities for preprocessing-based optimizations involving round-key matrices. The S-boxes operate on ordered triples of bits, and each S-box in a round can be evaluated (in parallel) using just three 1-bit multiplications a piece, leveraging their representation in algebraic normal form [1; Appendix C]:

$$(a, b, c) \mapsto (a \oplus bc, a \oplus b \oplus ac, a \oplus b \oplus c \oplus ab).$$

### Conditional swapping

The conditional swap functionality  $([\bar{x}], [\bar{y}]) \leftarrow F_{SWAP}([x], [y], [B])$  uses the standard trick for conditional branching in MPC. Servers 0 and 1 respectively hold  $(2, 2)$ -additive shares of (i) an ordered pair  $(x, y)$  and (ii) a selection bit  $B$  on which to condition the swap. They compute a sharing  $[z]$  of  $z = x + B(y - x)$ , and then output  $([\bar{x}], [\bar{y}]) = ([z], [x] + [y] - [z])$ . An easy calculation verifies that  $(\bar{x}, \bar{y}) = (x, y)$  if  $B = 0$  and  $(\bar{x}, \bar{y}) = (y, x)$  if  $B = 1$ . Figure 7.1 illustrates the conditional swap protocol.

### Non-private DPF traversal

Recall from Section 3.2 that Boyle–Gilboa–Ishai DPF keys have the form

$$\mathbf{dpf}[b] := (v_b^{(e)}, \overline{cw}[1], \dots, \overline{cw}[h]) \in (\mathbb{F}_{2^\lambda})^h \times \mathbb{F}_{2^L},$$

where  $h$  is the tree height. In any honestly generated DPF key pair, the sequence of CWs  $\overline{cw}[1], \dots, \overline{cw}[h]$  is *common to both keys* while the  $v_b^{(e)}$  are sampled independently at random (subject to having opposite least-significant bits). We therefore refer to key pairs having identical CW sequences as *plausible key pairs*, reflecting the fact that adjudicating their validity is more involved than simple plaintext matching among values held in common by the two shareholders.

For a given bit string  $i \in \{0, 1\}^*$  of length  $j < h$ , we write  $v_b^{(i||0)}$  and  $v_b^{(i||1)}$  to denote respectively the left and right children of node  $v_b^{(i)}$  in the tree share induced by  $\mathbf{dpf}[b]$ , so that the parenthesized superscript on any given node indicates the sequence of left and right traversals needed to arrive at that node starting from  $v_b^{(e)}$ . In the audit protocol, the servers will relate corresponding pairs  $(v_0^{(i)}, v_1^{(i)})$  with nodes  $v^{(i)} = v_0^{(i)} + v_1^{(i)}$  on a “reconstructed” tree. Definitionally, a given key pair is well-formed if and only if this reconstructed tree represents a generalized point function; the servers can appeal to [Bullet 3 of Corollary 1](#) (in [Section 2.7.1](#)) to check this.

To see how this works, we first look at (non-private) node traversal in Boyle–Gilboa–Ishai DPFs. To traverse from  $v_b^{(i)}$  (with  $|i| = j$ ) to its  $B$ th child,  $v_b^{(i||B)}$ , first parse  $v_b^{(i)}$  as  $(s_b^{(i)}, \cdot, \text{flag}_b^{(i)}) \in \mathbb{F}_{2^{\lambda-2}} \times \mathbb{F}_2 \times \mathbb{F}_2$ . From here:

1. compute  $(\text{child}_b^{(i||0)}, \text{child}_b^{(i||1)}) \leftarrow G_\lambda^{2\times}(s_b^{(i)} || 00)$ ;
2. parse  $\text{child}_b^{(i||B)}$  as  $(\overline{s}_b^{(i||B)}, \cdot, T_b^{(i||B)}) \in \mathbb{F}_{2^{\lambda-2}} \times \mathbb{F}_2 \times \mathbb{F}_2$  and  $\overline{cw}[j]$  as  $(\overline{\overline{cw}}^{(j)}, t_1^{(j)}, t_0^{(j)}) \in \mathbb{F}_{2^{\lambda-2}} \times \mathbb{F}_2 \times \mathbb{F}_2$ ;
3. compute  $s_b^{(i||B)} \leftarrow \overline{s}_b^{(i||B)} + (\overline{\overline{cw}}^{(j)} \cdot (1 + \text{flag}_b^{(i)}))$  and  $\text{flag}_b^{(i||B)} \leftarrow T_b^{(i||B)} + (t_B^{(j)} \cdot \text{flag}_b^{(i)})$ ; and then
4. output  $v_b^{(i||B)} := (s_b^{(i||B)}, \cdot, \text{flag}_b^{(i||B)}) \in \mathbb{F}_{2^\lambda} \times \mathbb{F}_2 \times \mathbb{F}_2$  as the child.<sup>1</sup>

Notice that whether or not to apply  $\overline{cw}[j]$  depends entirely on  $\text{flag}_b^{(i)}$ ; consequently, if corresponding nodes induced by a plausible key pair ever collide, then so too must all of their descendants. Recall from [Section 3.2](#) that we call such colliding-node pairs *0-pairs*; we likewise call *non-colliding* pairs *1-pairs* if either (i) they are at the leaf layer, or (ii) their children comprise both a 0-pair and a 1-pair. The next theorem is a direct consequence of these definitions

<sup>1</sup>For a comprehensive treatment of this construction, we defer to the original paper by Boyle et al. [[14](#); §3.2.2].

**Theorem 8.** Suppose  $(\mathbf{dpf}[0], \mathbf{dpf}[1])$  is a plausible key pair inducing  $(v_0^{(i)}, v_1^{(i)})$  and consider the node  $v^{(i)} := v_0^{(i)} + v_1^{(i)}$  in the reconstructed tree. The following both hold:

1.  $v^{(i)}$  is a 0-node if and only if  $(v_0^{(i)}, v_1^{(i)})$  is a 0-pair; and
2.  $v^{(i)}$  is a 1-node if and only if  $(v_0^{(i)}, v_1^{(i)})$  is a 1-pair.

Our  $(2 + 1)$ -party audit protocol combines [Theorem 8](#) with [Corollary 1](#) to check the well-formedness of a given (plausible) key pair: Given their respective keys together with a bitwise sharing of the (purported) distinguished input  $i$ , the servers (i) traverse to both children of the root using the above procedure, (ii) reconstruct the  $(1 - B^{(1)})$ th child to ensure it is a 0-node, and then (iii) recurse on the height- $(h - 1)$  tree rooted at the  $(B^{(1)})$ th child.

If any  $(1 - B^{(j)})$ th child along the path from the root to the leaf layer is not of type 0, then  $i$  does not define a 1-path in the reconstructed tree and the servers reject the key pair; otherwise, as per [Bullet 3](#) of [Corollary 1](#), the servers conclude that  $(\mathbf{dpf}[0], \mathbf{dpf}[1])$  constitutes a well-formed DPF with distinguished input  $i$ .

### MPC-based auditing

The  $(2+1)$ -party audit protocol implements the strategy just described via alternating applications of the oblivious PRG ([§7.2.1](#)) and conditional swap ([§7.2.1](#)) functionalities, woven together with some additional Du-Atallah multiplications ([§2.5](#)).

For each  $b = 0, 1$ , server  $b$  receives as input its DPF key  $\mathbf{dpf}[b] := (v_b^{(e)}, \overline{cw}[1], \dots, \overline{cw}[h])$  and a  $(2, 2)$ -additive share  $[i]_b := ([B^{(1)}]_b, \dots, [B^{(h)}]_b)$  of the (purported) distinguished input  $i$ . Server 2 assists with the computation but receives no input and produces no output.

To begin, servers 0 and 1 compare CW sequences to ensure they are auditing a plausible key pair; if so, server  $b$  uses its inputs and  $G_\lambda^{2 \times}$  to *non-obliviously* traverse to both children of the root; i.e., it computes

$$(v_b^{(0)}, v_b^{(1)}) := ((s_b^{(0)}, \cdot, \text{flag}_b^{(0)}), (s_b^{(1)}, \cdot, \text{flag}_b^{(1)}))$$

via Steps (1)–(4) from [Section 7.2.1](#), and then it secret shares both halves of the output with server  $(1 - b)$ .

Define  $i_0 = \varepsilon$  and, for each  $j = 1, \dots, h$ , define  $i_j = i_{j-1} || B^{(j)}$  and  $\bar{i}_j = i_{j-1} || (1 - B^{(j)})$ . (Thus,  $i_j$  is the length- $j$  prefix of  $i$  and  $\bar{i}_j$  is the length- $(j - 1)$  prefix of  $i$  followed

by an incorrect  $j$ th bit.) Upon receiving its shares from server  $(1 - b)$ , server  $b$  now holds both  $([v_0^{(i_b||0)}]_b, [v_0^{(i_b||1)}]_b)$  and  $([v_1^{(i_b||0)}]_b, [v_1^{(i_b||1)}]_b)$ .

Enlisting the help of server 2, for each  $j = 1, \dots, h$ , servers 0 and 1 then:

1. invoke  $F_{SWAP}$  for each  $b = 0, 1$  to produce
  - $([v_b^{(i_j)}], [v_b^{(i_j)}]) \leftarrow F_{SWAP}([v_b^{(i_{j-1}||0})], [v_b^{(i_{j-1}||1)}], [B^{(j)}]);$
2. reconstruct  $v^{(i_j)} := v_0^{(i_j)} + v_1^{(i_j)}$  and **reject** if  $v^{(i_j)} \neq 0$ ;
3. for each  $b = 0, 1$ , parse
  - $[v_b^{(i_j)}]$  as  $([\bar{s}_b^{(i_j)}], \cdot, [\text{flag}_b^{(i_j)}]);$
4. invoke  $G_\lambda$  for each  $b = 0, 1$  to produce
  - $([\text{child}_b^{(i_j||0)}], [\text{child}_b^{(i_j||1)}]) \leftarrow G_\lambda([\bar{s}_b^{(i_j)}]||00);$
5. for each  $b = 0, 1$  and  $B = 0, 1$ , parse
  - $[\text{child}_b^{(i_j||B)}]$  as  $([\bar{s}_b^{(i_j||B)}], \cdot, [T_b^{(i_j||B)}]);$
6. parse  $\overline{cw}[j+1]$  as  $(\overline{cw}^{(j+1)}, t_1^{(j+1)}, t_0^{(j+1)});$
7. for each  $b = 0, 1$  and  $B = 0, 1$ , compute
  - $[s_b^{(i_j||B)}] \leftarrow [\bar{s}_b^{(i_j||B)}] + (\overline{cw}^{(j+1)} \cdot (b + [\text{flag}_b^{(i_j)}])),$
  - $[\text{flag}_b^{(i_j||B)}] \leftarrow [T_b^{(i_j||B)}] + (t_B^{(j+1)} \cdot [\text{flag}_b^{(i_j)}]);$  and,
8. finally, for each  $b = 0, 1$  and  $B = 0, 1$ , set
  - $v_b^{(i_j||B)} \leftarrow ([s_b^{(i_j||B)}], \cdot, [\text{flag}_b^{(i_j||B)}]).$

They **accept** if they did not **reject** in Step 2 for any  $j$ .

We note that both functionalities  $F_{PRG}$  and  $F_{SWAP}$  consist exclusively of (perfectly simulatable) Du-Atallah multiplications and non-interactive linear operations (and are, therefore, trivial to simulate); moreover, with the sole exceptions of comparing CWs and reconstructing the  $v^{(i_j)}$  in Step 2, all remaining steps likewise consist exclusively of Du-Atallah multiplications and linear operations. The reconstruction in Step 2 yields no information unless  $v^{(i_j)} \neq 0$ , in which case the servers learn that the  $j$ th node along the purported 1-path is incorrect. It is easy to verify that if the key pair is indeed well-formed with distinguished input  $i$ , then the servers always



**accept**; conversely, if the key pair is not a DPF with distinguished input  $i$ , then the existence of some prefix  $i_j$  of  $i$  such that  $v^{(i_j)}$  is not a 0-node follows contrapositively from [Corollary 1](#). In this case, the servers will **reject** in Step 2 when traversing from  $v^{(i_{j-1})}$  to  $v^{(i_j)}$ . We have thus proved the following theorem.

**Theorem 9.** *The  $(2 + 1)$ -party auditing protocol is complete and perfectly sound. Moreover, the view of any semi-honest party in an accepting run of the  $(2 + 1)$ -party auditing protocol is perfectly simulatable.*

### 7.3 MPC-in-the-head ZKPok verification

The preceding section described a  $(2 + 1)$ -party computation using which servers 0 and 1 can (with assistance from a semi-honest server 2) efficiently check the well-formedness of their respective Boyle–Gilboa–Ishai DPF keys. We now explain the conversion of this  $(2 + 1)$ -party computation into a zero-knowledge argument based on MPC-in-the-head.

The high-level idea is for the prover, who holds  $\mathbf{dpf}[0]$  and  $\mathbf{dpf}[1]$  and the distinguished input  $i$ , to *simulate* the  $(2 + 1)$ -party audit and then commit to the (unidirectional) communication channels over which the servers exchange messages in the simulation. From here, the verifiers challenge the prover to open subsets of these commitments for inspection.

#### MPC-in-the-head with multiple verifiers

A critical difference between our arguments and other zero-knowledge arguments constructed in the MPC-in-the-head paradigm stems from (i) our use of a  $(2 + 1)$ -party computation secure against a *single passive corruption*, combined with the fact that (ii) the private inputs to servers 0 and 1 in the underlying MPC include the private inputs to servers 0 and 1 in Sabre (i.e.,  $\mathbf{dpf}[0]$  and  $\mathbf{dpf}[1]$ , respectively). Consequently, server  $b$  can scrutinize the view of *either* server  $b$  or server 2 from a given simulation, but *never both*—and it can *never* scrutinize the view of server  $(1 - b)$ . This fact all but rules out sound “single-verifier” arguments, as a cheating prover could always, e.g., confine inconsistencies to transcripts that the verifier is not allowed to scrutinize.

We circumvent this issue by leveraging two or more verifiers that each examine a different subset of transcripts. The argument is accepted if and only if *all* verifiers (i) receive identical channel commitments from the prover, and (ii) find no inconsistencies in the views they scrutinize. Definitionally, such “multi-verifier” MPC-in-the-head arguments are equivalent to *secret-shared non-interactive proofs* (SNIPs) with a (modestly) generalized zero-knowledge property: whereas Corrigan-Gibbs and Boneh’s SNIP definition [20; §4.1] insists that the view of *any proper subset* of verifiers be simulatable, “multi-verifier MPC-in-the-head” SNIPs inherit their simulatability requirements from the access structures governing privacy in the underlying MPC. In the case of Sabre’s  $(2 + 1)$ -party auditing, the requirement is that the view of *any given verifier* be simulatable. When there are just two verifiers, this requirement coincides with the one originally proposed by Corrigan-Gibbs and Boneh.

### 3-verifier SNIP auditing

The 3-verifier SNIP audit protocol employs  $(2 + 1)$ -party auditing while eliminating the need for (almost all) interaction among the servers. Specifically, upon sampling its DPF keys, the prover simulates a single run of the  $(2+1)$ -party audit protocol from [Section 7.2.1](#), committing to the ordered sequence of messages on each (unidirectional) simulated communication link among the three parties. Let  $M^{a \rightarrow b}$  denote the ordered sequence of messages sent from party  $a$  to party  $b$  in the simulation, and let  $H^{a \rightarrow b} := \text{Hash}(M^{a \rightarrow b})$  for some cryptographic hash function  $\text{Hash}: \{0, 1\}^* \rightarrow \{0, 1\}^\mu$  (our implementation uses SHA256 truncated to  $\mu = 128$  bits).

Notice that  $(\text{dpf}[0], M^{1 \rightarrow 0}, M^{2 \rightarrow 0})$  suffices to reconstruct the view of server 0, while  $(\text{dpf}[1], M^{0 \rightarrow 1}, M^{2 \rightarrow 1})$  suffices to reconstruct the view of server 1 and  $(M^{2 \rightarrow 0}, M^{2 \rightarrow 1})$  constitutes the entire view of server 2. The prover constructs a 4-ary Merkle-tree (with height 1) having root  $\mathbf{H} := \text{Hash}(\overline{cw}[0] || \dots || \overline{cw}[h] || H^{0 \rightarrow 1} || H^{1 \rightarrow 0} || H^{2 \rightarrow 0} || H^{2 \rightarrow 1})$ , and then it sends

1.  $\Pi^{(0)} := (M^{1 \rightarrow 0}, M^{2 \rightarrow 0}, H^{2 \rightarrow 1})$  to server 0,
2.  $\Pi^{(1)} := (M^{0 \rightarrow 1}, M^{2 \rightarrow 1}, H^{2 \rightarrow 0})$  to server 1, and
3.  $\Pi^{(2)} := (H^{0 \rightarrow 1}, H^{1 \rightarrow 0}, M^{2 \rightarrow 0}, M^{2 \rightarrow 1})$  to server 2.

To verify its portion of the SNIP, server 0 uses  $(\mathbf{dpf}[0], M^{1 \rightarrow 0}, M^{2 \rightarrow 0})$  to recreate  $M^{0 \rightarrow 1}$ ; it rejects if  $(M^{0 \rightarrow 1}, M^{1 \rightarrow 0}, M^{2 \rightarrow 0}, H^{2 \rightarrow 1})$  is not consistent with the Merkle root  $\mathbf{H}$ . Servers 1 and 2 verify their portions analogously. The 3-verifier SNIP is accepting if and only if (i) each of the three servers received the same Merkle root  $\mathbf{H}$ , and (ii) none of the servers rejects its portion of the SNIP. Since the three verifiers collectively scrutinize all views from the simulation, *inconsistencies in the simulation cannot escape notice by at least one verifier* (discounting the negligible probability of hash collisions when constructing the Merkle tree). In particular, we have just argued that—except with probability negligible in the hash-length  $\mu$ —a prover can produce an accepting 3-verifier SNIP for  $(\mathbf{dpf}[0], \mathbf{dpf}[1])$  and  $[i]$  only if these same values would have passed  $(2 + 1)$ -party auditing—which has perfect soundness. Moreover, because each verifier merely inspects one view from the  $(2 + 1)$ -party audit protocol, the view of any given verifier remains trivially simulatable; we have thus proved the following theorem.

**Theorem 10.** *The 3-verifier SNIP auditing protocol with hash function  $\text{Hash}: \{0, 1\}^* \rightarrow \{0, 1\}^\mu$  is perfectly simulatable and has perfect completeness and soundness overwhelming in  $\mu$ .*

A more detailed proof sketch for [Theorem 10](#) is included as [Appendix C](#).

## 2-verifier SNIP auditing

The 2-verifier SNIP audit protocol employs cut-and-choose to eliminate the need for a third verifier while maintaining soundness error negligible in  $\mu$ . Specifically, upon sampling its DPF keys, the prover now runs  $\mu$  parallel simulations of the  $(2 + 1)$ -party audit protocol, committing to each unidirectional communication link among the three parties in each simulation. For each  $i = 1, \dots, \mu$ , let  $M_i^{a \rightarrow b}$  denote the ordered sequence of messages sent from party  $a$  to party  $b$  in the  $i$ th parallel simulation and let  $H_i^{a \rightarrow b} := \text{Hash}(M_i^{a \rightarrow b})$ .

As in the 3-verifier SNIPs, the prover constructs a Merkle tree committing to all unidirectional channel commitments. In particular, for each  $i = 1, \dots, \mu$ , it computes the digest  $\mathbf{H}_i := \text{Hash}(\overline{\text{cw}}[0] || \dots || \overline{\text{cw}}[h] || H_i^{0 \rightarrow 1} || H_i^{1 \rightarrow 0} || H_i^{2 \rightarrow 0} || H_i^{2 \rightarrow 1})$ , and then it constructs the Merkle root as  $\text{Hash}(\mathbf{H}_1 || \dots || \mathbf{H}_\mu)$ . Regarding  $\text{Hash}$  as a random

oracle, each of the  $\mu$  bits of the Merkle root constitutes a distinct “challenge” (à la Fiat–Shamir [26]): when the bit is 0, each verifier will inspect “its own” view; when the bit is 1, both verifiers will inspect server 2’s view. Such a tree is depicted in Figure 7.2.

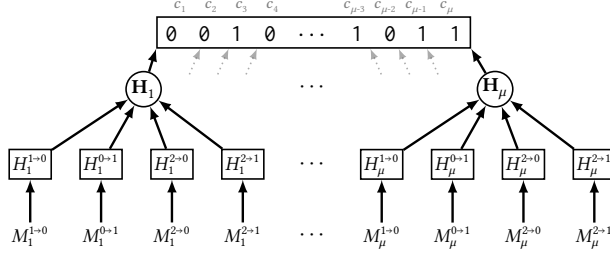


Figure 7.2: Merkle-tree commitment for 2-verifier SNIPs

Let  $c_i$  denote the  $i$ th bit of the Merkle root. For each  $i = 1, \dots, \mu$ , the prover discloses the tuple

$$\Pi_i^{(0)} := \begin{cases} (M_i^{1 \rightarrow 0}, M_i^{2 \rightarrow 0}, H_i^{2 \rightarrow 1}) & \text{if } c_i = 0, \text{ and} \\ (M_i^{2 \rightarrow 0}, M_i^{2 \rightarrow 1}) & \text{otherwise,} \end{cases}$$

to server 0, and symmetrically for server 1.

To verify its portion of the SNIP, server  $b$  uses  $\mathbf{dpf}[b]$  and the  $\Pi_i^{(b)}$  to reconstruct the missing leaf hashes, and then it checks their consistency with the Merkle root. The 2-verifier SNIP is accepting if and only if (i) both servers received the same Merkle root  $\mathbf{H}$ , (ii) the tuple disclosed in each  $\Pi_i^{(b)}$  is consistent with the challenge bit  $c_i$  from this root, and (iii) neither server rejects its portion of the SNIP.

As per Theorem 10, even a single inconsistency-free simulation suffices to establish well-formedness of the DPF with probability overwhelming in  $\mu$ ; moreover, if a given simulation *does* have an inconsistency, then this inconsistency must be evident in the view of either server 2 or at least one of servers 0 or 1 and will, therefore, be detected with probability at least  $\frac{1}{2}$ . Furthermore, as with 3-verifier SNIP auditing, the view of any given verifier remains trivially simulatable. Hence, we obtain the following theorem.

Table 7.1: Communication cost of auditing (in bytes) for a client in a Sabre instance with  $n = 2^h$  buckets/mailboxes. In the table,  $h = \lg n$  denotes the height of the DPF tree, while  $r$  and  $s$  respectively denote the number of rounds and S-boxes in LowMC with 128-bit blocks. The 2-verifier SNIP employs cut-and-choose with soundness error  $\approx 2^{-128}$ .

Audit type	client $\rightarrow$ server $b$	client $\rightarrow$ server 2
(2+1)-party	—	—
3-verifier SNIP	$112 + \lceil (64.25 + 3sr)(h-1) \rceil$	64
2-verifier SNIP	$37\,888 + (4\,112 + 192sr)(h-1)$	—

Table 7.2: Communication cost of auditing (in bytes) for the servers in a Sabre instance with  $n = 2^h$  buckets/mailboxes. As in Table 7.1,  $h = \lg n$  denotes the height of the DPF tree, while  $r$  and  $s$  respectively denote the number of rounds and S-boxes in LowMC with 128-bit blocks. Similarly, the 2-verifier SNIP employs cut-and-choose with soundness error  $\approx 2^{-128}$  just as in Table 7.2.

Audit type	server $b \leftrightarrow$ server $(1-b)$	server $b \rightarrow$ server 2	server 2 $\rightarrow$ server $b$
(2+1)-party	$16 + (16\frac{1}{8} + \frac{3}{2}sr)(h-1)$	$16 + (16\frac{1}{8} + \frac{3}{2}sr)(h-1)$	$16 + (32\frac{1}{8} + \frac{3}{2}sr)(h-1)$
3-verifier SNIP	16	16	16
2-verifier SNIP	16	—	—

**Theorem 11.** *The 2-verifier SNIP auditing protocol with hash function  $\text{Hash}: \{0, 1\}^* \rightarrow \{0, 1\}^\mu$  is perfectly simulatable and has perfect completeness and soundness overwhelming in  $\mu$ .*

A more detailed proof sketch for Theorem 11 is included as Appendix D.

## 7.4 DPF key generation in MPC

The MPC protocol outlined in Section 7.2.1 works by traversing down the 1-path of a DPF tree. We can observe that this traversal is also a central step when generating keys for a Boyle–Gilboa–Ishai DPF which suggests that this protocol can be modified to generate new DPF keys rather than just verifying existing keys. Just like the verification protocol, the key generation protocol relies upon alternating applications of the oblivious PRG and conditional swap functionalities from Section 7.2.1. The protocol to generate DPF keys is shown in Algorithm 2.

**Algorithm 2** MPC DPF Key Generation**Require:** Party  $b \in \{0, 1\}$ **Require:**  $(\langle x \rangle)_b$ 

$$s_b^{(i_0)} \in_R \mathbb{Z}_{2^\lambda}$$

$$[t_0^{(0)}]_b \in_R \{0, 1\}$$

$$[t_1^{(0)}]_b = [t_0^{(i_0)}]_b \oplus b$$

**for all**  $j = 1$  **to**  $n$  **do**  **if**  $j = 1$  **then**    Locally compute:  $([\text{child}_b^{(i_{j-1}||0)}]_b, [\text{child}_b^{(i_{j-1}||1)}]_b) = G_\lambda([s_b^{(j-1)}]||00)$ 

$$([\text{child}_b^{(i_{j-1}||0)}]_b, [\text{child}_b^{(i_{j-1}||1)}]_b) = (0^\lambda, 0^\lambda)$$

**else**    MPC compute:  $([\text{child}_c^{(i_{j-1}||0)}]_b, [\text{child}_c^{(i_{j-1}||1)}]_b) = G_\lambda([s_c^{(j-1)}]||00)$  for  $c \in \{0, 1\}$   **end if**

$$([\text{child}_c^{(i_j)}]_b, [\text{child}_c^{(i_j)}]_b) \leftarrow F_{SWAP}([\text{child}_c^{(i_{j-1}||0)}]_b, [\text{child}_c^{(i_{j-1}||1)}]_b, [x^{(j)}]) \text{ for } c \in \{0, 1\}$$

  Parse  $[\text{child}_c^{(i_j)}]_b$  as  $([\bar{s}_c^{(i_j)}]_b, \cdot, [T_c^{(i_j)}]_b)$  for  $c \in \{0, 1\}$   Parse  $[\text{child}_c^{(i_j)}]_b$  as  $([\bar{s}_c^{(i_j)}]_b, \cdot, [T_c^{(i_j)}]_b)$  for  $c \in \{0, 1\}$ 

$$[\bar{c\mathbf{w}}]_b = [\bar{s}_0^{(i_j)}]_b \oplus [\bar{s}_1^{(i_j)}]_b$$

$$[t_0^{(j)}]_b = [T_0^{(i_{j-1}||0)}]_b \oplus [T_1^{(i_{j-1}||0)}]_b \oplus [x^{(j)}]_b \oplus 1$$

$$[t_1^{(j)}]_b = [T_0^{(i_{j-1}||1)}]_b \oplus [T_1^{(i_{j-1}||1)}]_b \oplus [x^{(j)}]_b$$

$$[\bar{c\mathbf{w}}^{(j)}]_b = ([\bar{c\mathbf{w}}]_b, [t_0^{(j)}]_b, [t_1^{(j)}]_b)$$

$$[s_0^{(j)}]_b = [\bar{s}_0^{(i_j)}]_b \oplus [\bar{c\mathbf{w}}]_0 \cdot [t_0^{(j-1)}]_b$$

$$[t_0^{(j)}]_b = [T_0^{(i_j)}]_b \oplus ([T_0^{(i_j)}]_b \oplus [T_0^{(i_j)}]_b \oplus 1) \cdot [t_0^{(j-1)}]_b$$

$$[s_1^{(j)}]_b = [\bar{s}_1^{(i_j)}]_b \oplus [\bar{c\mathbf{w}}]_1 \cdot [t_1^{(j-1)}]_b$$

$$[t_1^{(j)}]_b = [T_1^{(i_j)}]_b \oplus ([T_1^{(i_j)}]_b \oplus [T_1^{(i_j)}]_b \oplus 1) \cdot [t_1^{(j-1)}]_b$$

**end for**

$$[\bar{c\mathbf{w}}^{(n+1)}]_b = y \oplus H(s_0^{(n)}) \oplus H(s_1^{(n)})$$

$$[k'_c]_b = ([t_c^{(0)}]_b, [\bar{c\mathbf{w}}^{(1)}]_b, \dots, [\bar{c\mathbf{w}}^{(n+1)}]_b) \text{ for } c \in \{0, 1\}$$

  Send  $[k'_b]_b$  to party  $P_{\bar{b}}$   Receive  $[k'_b]_{\bar{b}}$  from party  $P_{\bar{b}}$ 

$$\text{Reconstruct } k_b = s_b^{(i_0)} || ([k'_b]_b \oplus [k'_b]_{\bar{b}}) = (s_b^{(i_0)}, t_c^{(0)}, \bar{c\mathbf{w}}^{(1)}, \dots, \bar{c\mathbf{w}}^{(n+1)})$$

**Return**  $k_b$

In this protocol,  $H$  denotes a PRG, which produces DPF tree leaves from the desired output group  $\mathbb{G}$ .

The precomputed values for this computation can come from either the server in a  $(2 + 1)$ -party protocol or can be precomputed in a pure 2-party protocol. In practice,  $(2 + 1)$ -party protocols will generally favour the technique of DPF rotation explained for selection vectors in [Section 2.6](#) as this requires only a single round of communication between the two client parties rather than  $O(\lg n)$  rounds. However, in a 2-party setting there is no server party to provide a pre-generated DPF with a randomly chosen distinguished point. As a result, the logarithmic protocol for MPC generation of DPF keys allows GROTTO to be used, relatively efficiently, in a 2-party setting.

## 7.5 Sabre sender-anonymous messaging

We present Sabre, a family of SAM protocols with instances operating in both the bulletin board and mailbox models. Sabre protocols inherit much of their basic structure from Riposte and Express, but incorporate key innovations that improve not only concrete performance and scalability under normal operations, but also resilience to resource exhaustion-style DoS attacks in the mailbox model.

From a technical perspective, the primary difference is how Sabre implements auditing to identify malformed write requests: the *senders* construct compact (2- or 3-verifier) SNIPs that “directly” attest to the well-formedness of the DPFs they submit. We construct the SNIPs in a novel paradigm we call *multi-verifier MPC-in-the-head*, which, as discussed in [Section 7.3](#), generalizes the (single-verifier) MPC-in-the-head paradigm of Ishai, Kushilevitz, Ostrovsky, and Sahai [39] to achieve good soundness at low cost in scenarios where two or more distinct verifiers can check two or more distinct subsets of simulated interaction transcripts. In the sender-anonymous mailbox model, Sabre also uses an additional trick to decouple the cost of DPF evaluation from the bit length of the mailbox addresses. Together, these modifications yield speedups that exceed an order of magnitude relative to Express when all the write requests are “well-formed” and increase significantly—with the performance gap growing exponentially in the number of mailboxes—in the presence of resource-exhaustion DoS attacks.

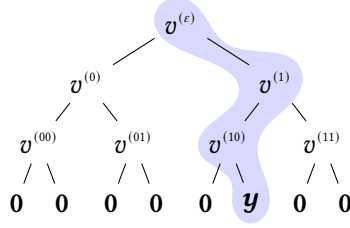


Figure 7.3: Binary-tree representation of the generalized point function with point  $(x_5, y) \in D \times R$ ,  $|D| = 2^3$ .

As with both Riposte and Express, the *correctness* of Sabre requires that all servers faithfully audit all incoming DPFs—that is, none of these protocols can operate in the presence of Byzantine servers. However, like Express, Sabre’s reliance on non-interactive (SNIP-based) auditing deprives would-be malicious servers of the opportunity to deviate from the protocol in ways that might leak information about the mapping between writers and the messages they have penned.

### 7.5.1 System Design

This section presents the system design of Sabre in both the sender-anonymous bulletin board model (Sabre-BB) and the sender-anonymous mailbox model (Sabre-M). [Appendix E](#) describes four distinct variants of Sabre-M: One that closely mimics the design of Express; a second that adaptively modifies the mapping between DPFs and mailbox addresses to improve efficiency; and a third and fourth that push the optimization in the second variant to its logical extreme. The first three variants serve as “stepping stones” toward Sabre-M (which is what we describe in the main text).

Each of Sabre-BB and Sabre-M is instantiable in three ways (deriving from three ways to instantiate the audit protocol); namely, as (i) a 2-server version using “2-verifier MPC-in-the-head”, (ii) a 3-server version using “3-verifier MPC-in-the-head”, or (iii) a 3-server version directly using secure  $(2 + 1)$ -party computation. Except where otherwise specified, all comparisons of Sabre’s performance to those of Riposte and Express refer to a 2-server instantiation—which provides the *strongest security guarantees* but the *poorest performance* of the three options. That said, we



begin with a high-level description that treats auditing as an inscrutable black box, abstracting away the details described in [Section 7.2](#). The reader should bear in mind that Sabre auditing requires just  $O(\lambda \lg n)$  work to the  $\Omega(\lambda n)$  work required by Riposte and Express auditing, and that Sabre auditing renders judgement *before* the servers ever “evaluate” their DPF keys (a linear-cost operation). Indeed, Sabre’s improved DoS resistance stems from Sabre servers’ ability to rapidly audit incoming DPF keys.

### Sabre for sender-anonymous bulletin boards

Consider a (2- or 3-server) Sabre-BB instance with security parameter  $\lambda \in \mathbb{N}$  (say,  $\lambda = 128$ ) and a bulletin board comprising  $n = 19.5m$  buckets each capable of holding a single  $L$ -bit message. Thus, we have domain  $D = [0 \dots n - 1]$  and range  $R = \mathbb{F}_{2^L}$ . The design of Sabre-BB tightly parallels that of Riposte, save for the adoption of Boyle et al.’s more compact DPFs and the new audit protocol: A sender who wishes to post a message  $M \in \mathbb{F}_{2^L}$  to the bulletin board

1. samples a random bucket index  $i \in_R D$ ;
2. samples  $(\mathbf{dpf}[0], \mathbf{dpf}[1]) \leftarrow \text{Gen}(1^\lambda, D, R; i, M)$ ; and then
3. sends  $\mathbf{dpf}[b]$  to server  $b$  for  $b = 0, 1$ .

Upon receiving and auditing  $\mathbf{dpf}[b]$ , server  $b$  constructs the vector  $M_b \in (\mathbb{F}_{2^L})^{1 \times n}$  in which the  $j$ th component equals  $\text{Eval}(\mathbf{dpf}[b], j)$  for  $j = 0, \dots, n - 1$ ; server  $b$  adds  $M_b$  to its bulletin board database to effectuate the write.<sup>2</sup> We remark that  $M_b$  is efficiently computable (with a cost dominated by  $n - 1$  length-doubling PRG evaluations and  $n$  evaluations of the leaf-stretching PRG) using the so-called *full-domain evaluation* procedure described by Boyle et al. [[14](#); §3.2.3].

As per [Definition 4](#),  $M_0 + M_1 \in (\mathbb{F}_{2^L})^{1 \times n}$  has message  $M$  in its  $i$ th column and zero elsewhere—*provided the sender generated  $(\mathbf{dpf}[0], \mathbf{dpf}[1])$  honestly*, which the servers confirm via any of the three audit protocol instantiations.

<sup>2</sup>As in Riposte, Sabre-BB writers can employ Newton sums, writing pairs  $(M, M^2)$  to hedge against inevitable 2-way collisions.

### Sabre for sender-anonymous mailboxes

We now describe Sabre-M, the most performant variant of Sabre in the mailbox model. (Recall that we describe three additional “stepping-stone” variants in [Appendix E](#).) Consider a (2- or 3-server) Sabre-M instance with security parameter  $\lambda \in \mathbb{N}$  (say,  $\lambda = 128$ ) and  $n$  mailboxes each capable of holding a single  $L$ -bit message. (Thus, we have domain  $D = [0 \dots n - 1]$  and range  $R = \mathbb{F}_{2^L}$ ). The design of Sabre-M closely follows that of Express.

*Mailbox registration:* As with Express, prospective recipients must pre-register a mailbox with the servers. In Sabre-M, the servers assign mailbox addresses deterministically using a pseudorandom function (PRF)  $F: \{0, 1\}^\lambda \times D \rightarrow \{0, 1\}^\lambda$ ; specifically, to register the  $i$ th mailbox, the servers compute  $addr_i \leftarrow F(\tilde{k}, i)$  using a long-lived secret key  $\tilde{k} \in \{0, 1\}^\lambda$  held by the servers, and then they return  $(i, addr_i)$  to the registrant.

*Writing:* To write a message  $M \in \mathbb{F}_{2^L}$  to the  $i$ th mailbox, the sender must know  $(i, addr_i)$ . Armed with this pair, it

1. samples  $(\mathbf{dpf}[0], \mathbf{dpf}[1]) \leftarrow \text{Gen}(1^\lambda, D, R; i, M)$ ;
2. samples additive shares  $[i]$  and  $[addr_i]$  of the distinguished input  $i$  and mailbox address  $addr_i$ ; and then
3. sends  $(\mathbf{dpf}[b], [i]_b, [addr_i]_b)$  to server  $b$  for  $b = 0, 1$ .

Notice that the distinguished input to the DPF is  $i \in [1 \dots n]$ , in contrast to  $addr_i \in \{0, 1\}^\lambda$  as used in Express. The discussion and experiments in [Appendix E](#) highlight the performance benefits of this difference.

Upon receiving and auditing  $(\mathbf{dpf}[b], [i]_b, [addr_i]_b)$ , server  $b$  expands  $\mathbf{dpf}[b]$  to the vector  $M_b \in (\mathbb{F}_{2^L})^{1 \times n}$  using the full-domain evaluation procedure [[14](#); §3.2.3], and then it adds  $M_b$  to its mailbox database to effectuate the write.

*Auditing:* Auditing in Sabre-M comprises two distinct steps, namely (i) checking that  $(\mathbf{dpf}[0], \mathbf{dpf}[1])$  is a well-formed DPF key pair (having distinguished input  $i$ ), and (ii) verifying that  $[addr_i]$  correctly shares the  $i$ th mailbox address.

For the former step (i.e., auditing the DPFs), Sabre-M uses any one of the three versions described in [Section 7.2](#); for the latter step (i.e., verifying the mailbox address) it uses a constant-complexity protocol described in [Section 7.5.2](#). Thus,

the entire auditing procedure has complexity logarithmic in  $n$ , allowing the servers to rapidly reject bogus write requests from a would-be resource-exhaustion DoS attacker. The benefits of rapid auditing are evident in our experimental evaluation in [Section 7.6](#).

### 7.5.2 Verifying Mailbox Addresses

This section describes how Sabre-M servers verify that incoming requests target valid mailbox addresses. The idea is quite simple: Recall that, along with  $\mathbf{dpf}[b]$ , the sender submits shares  $[i]_b$  and  $[addr_i]_b$  of the distinguished input  $i$  and the associated mailbox address  $addr_i$ . Also recall that the servers hold in common a secret key  $\tilde{k} \in \{0, 1\}^\lambda$  for a PRF  $F: \{0, 1\}^\lambda \times D \rightarrow \{0, 1\}^\lambda$ . Intuitively,  $F$  defines a deterministic mapping from the distinguished inputs  $i \in D$  to pseudorandom mailbox addresses  $addr_i := F(\tilde{k}; i)$ ; thus, the servers can use  $[i]$  and their knowledge of  $\tilde{k}$  to verify that  $[addr_i]$  shares the “correct” mailbox address.

To this end, the servers input  $[i]$  to  $F$  to compute a fresh sharing  $[addr'_i]$ , after which they hold two independent sharings: one,  $[addr_i]$ , of the *sender-claimed* address and another,  $[addr'_i]$ , of the *server-computed* address. Server  $b$  computes  $[A]_b := [addr_i]_b + [addr'_i]_b$  for  $b = 0, 1$ , and then the pair check if

$$\begin{aligned} A &= [A]_0 + [A]_1 \\ &= ([addr_i]_0 + [addr'_i]_0) + ([addr_i]_1 + [addr'_i]_1) \\ &= ([addr_i]_0 + [addr_i]_1) + ([addr'_i]_0 + [addr'_i]_1) \\ &= addr_i + addr'_i \\ &= \mathbf{0}, \end{aligned}$$

where (because arithmetic is in  $\mathbb{F}_{2^\lambda}$ ) the last equality holds if and only if  $addr_i = addr'_i$ . In particular, the check succeeds if and only if  $A_0 = A_1$ , which the servers can confirm alongside the DPF-verification portion of the audit protocol. Performing address checking early—before incurring the  $O(\lg n)$  cost of DPF verification—ensures that Sabre-M servers can always reject malicious write requests faster than a would be DoS attacker can produce them (see [Section 7.6.3](#)).

### 7.5.3 Security Guarantees

We define security for Sabre in the ideal-world/real-world simulation paradigm. In the ideal world, Sabre users hand their read and write requests directly to some ideal functionality, who faithfully executes the requested actions while leaking no superfluous information to external observers. In the real world, one of the 2- or 3-server Sabre-BB or Sabre-M instantiations replaces the ideal functionality. We then consider an attacker  $\mathcal{A}$  who controls an arbitrary number of readers and writers in addition to (at most) one server.

Informally, we wish to show that  $\mathcal{A}$  cannot exploit its privileged position as a Sabre server to compromise sender anonymity. We do this by exhibiting an efficient simulator that interacts with the ideal functionality to sample “simulated” views from a distribution close to the one describing  $\mathcal{A}$ ’s view in the real world. We then ask whether  $\mathcal{A}$  can adaptively conjure up sequences of events allowing it to distinguish between real and simulated views; if not, we conclude that the real Sabre protocols leak essentially nothing beyond what is leaked by their ideal-world counterparts.

Due to space constraints, we defer a detailed security definition and analysis to [Appendix F](#).

## 7.6 Implementation and Evaluation

To assess the practicality of Sabre, we wrote a proof-of-concept reference implementation in C++. Our implementation uses Boost.Asio v1.18.1 for asynchronous communication, OpenSSL 1.1.1i for hashing and TLS support, and dpf++ [36] for (2, 2)-DPFs; we wrote all other non-STL functionality by hand.<sup>3</sup>

**Experimental setup** We ran a series of experiments on Amazon EC2, with the servers running in geographically distant locations to mimic realistic Internet latency. The servers are all m5.4xlarge instances equipped with 64 GiB of RAM and 16 vCPUs, running the standard Ubuntu 18.04 AML. For each of the experiments reported in

---

<sup>3</sup>Our source code is available under the GNU General Public License (version 3) via <https://pr.iva.cy/sabre>.

this section, we configured LowMC to use 128-bit blocks (and 128-bit keys) with  $r = 19$  rounds consisting of  $s = 32$  S-boxes apiece. [Appendix G](#) presents experimental results to justify these parameter choices. (As a spoiler, this setting turns out to be pessimal with respect to  $(2 + 1)$ -party auditing and SNIPs, but optimal for the full-domain evaluation, which dominates the execution time.) We ran all experiments for 100 trials and report in our plots the sample mean over those 100 runs. (The plots also show error bars, but in most cases they are too small to see.)

In order to sustain high throughput, Sabre uses a custom *bitsliced* implementation of LowMC that operates on either 128 or 256 ciphertexts in parallel using SIMD operations; thus, several of our plots report the wall-clock time to process batches of size 128 where the reader might naturally expect to find the cost for “singleton” batches.

### 7.6.1 Communication Cost

We first present an (analytically determined) accounting of communication costs for all three variants of Sabre auditing. The costs are summarized in [Table 7.1](#) and [Table 7.2](#), with the derivations of these numbers included as [Appendix H](#). (Note that these costs account for *auditing only*; they do not include the DPF itself.) Observe that the client-to-server communication cost in both 2- and 3-verifier SNIP-based auditing scales with the product  $rs$  of LowMC parameters. (To a first approximation,  $r \approx \lg s$  so that proof-size scaling is more or less softly linear in  $r$ .) Also notice that 3-verifier SNIPs reduce communication about  $(\lambda/2)$ -fold relative to 2-verifier SNIPs; however, this reduction comes at the cost of an additional server and a correspondingly stronger non-collusion assumption.

### 7.6.2 Auditing

Our first set of experiments measure the cost of Sabre auditing and compare it with that of Riposte and Express auditing. [Figure 7.4](#) shows the relative cost of  $(2 + 1)$ -party, 3-verifier SNIP, and 2-verifier SNIP auditing in Sabre.

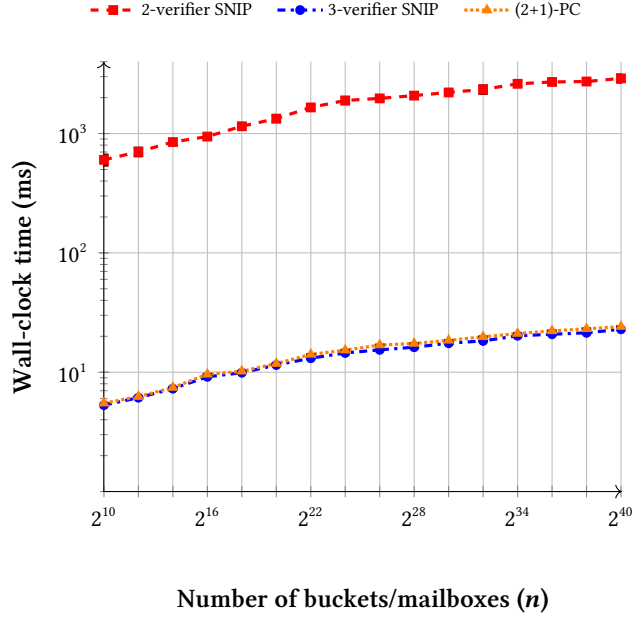


Figure 7.4: Time to audit a batch of 128 requests in Sabre

Owing to its reliance on cut-and-choose with  $\lambda = 128$  parallel instances, the 2-verifier SNIP auditing is by far the slowest of the pack, taking two orders of magnitude longer than both (2 + 1)-party and 3-verifier SNIP auditing. Nevertheless, the amortized cost per audit still barely exceeds 10 ms for Sabre instances even with  $n = 2^{20}$  registered mailboxes—indeed, even with up to  $n = 2^{40}$  mailboxes it never exceeds 23 ms. Although 2-verifier SNIP auditing is dramatically less performant than the alternatives, it offers superior security guarantees and the most apt comparison with Express; thus, we restrict our remaining experimental results to this variant. It is worth noting, however, that either 3-server variant would provide (significantly) better auditing performance wherever an additional party (and the resulting stronger trust assumption) is palatable. The 3-server variants also provide a more direct comparison with Riposte.

Figures 7.5a and 7.5b compare the auditing costs of Sabre with 2-verifier SNIP auditing versus those of Riposte and Express. These plots clearly illustrate the benefits of Sabre’s logarithmic auditing relative to Riposte’s and Express’ linear auditing. While the latter perform better for very small DPFs, the former becomes significantly faster as DPFs begin to exceed around a hundred thousand mailboxes

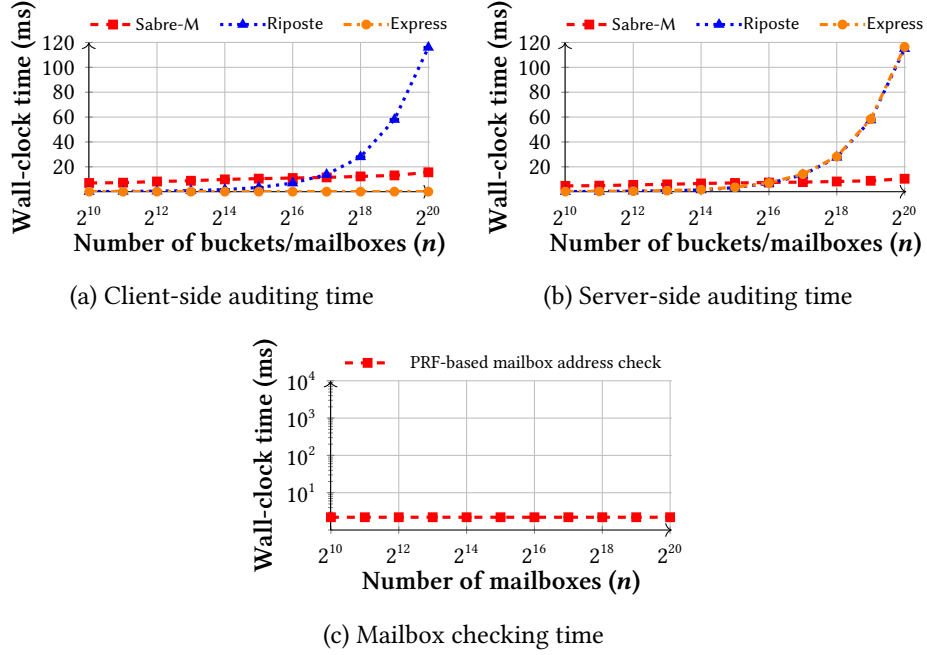


Figure 7.5: Sabre 2-verifier SNIP auditing versus Riposte and Express auditing. Observe that the  $y$ -axis in 7.5a and 7.5b are linear, while  $y$ -axis in 7.5c is log-scaled. Note that 7.5c reports mailbox-checking times for a batch of 128 requests.

(or buckets).

Figure 7.5c shows the cost of verifying a batch of 128 mailbox addresses in Sabre-M. As expected, the running time of the mailbox address check is constant (i.e., independent of the number mailboxes).

### 7.6.3 Resistance to denial-of-service attack

The next set of experiments characterizes Sabre’s resistance to resource-exhaustion DoS attacks by comparing how long it takes a simulated attacker to *produce* plausible-looking, yet malformed write requests with how long it takes Sabre auditing to *reject* them. The attacker in these experiments produces its malformed requests by sampling a sufficient quantity of pseudorandom bits from `/dev/urandom`. We consider two kinds of auditing failures, namely (i) when the address check fails and

(ii) when the address check passes but the DPF audit subsequently fails. (For the latter, we also consider at which tree depth the auditing aborts.)

Figure 7.6a plots our findings for the first kind of failure, comparing the time required to sample a *single* malformed request against the time required to check a *batch of 128 addresses* using the PRF-based address check. For Sabre instances with as few as  $n = 2^{10}$  mailboxes, sampling a malformed request takes nearly  $1.75\times$  as long as address checking; by  $2^{20}$  mailboxes the difference approaches  $4.6\times$ .

Figure 7.6b plots our findings for the second kind of failure, comparing the time required to construct a request that verifies up to but not including the  $d$ th level of the DPF (after which the attack samples the rest of the request from `/dev/urandom`) against the time required for the auditors to reject the request. Owing to SNIP verification’s probabilistic nature, the auditors can reject a malformed DPF upon inspecting only a fraction of the bits that the attacker must sample. As a result, sampling a malformed request takes about  $1.5\times$  as long as rejecting it via auditing, for all  $d$ .

In contrast to Riposte and Express auditing—where auditors run asymptotically slower than writers—Sabre auditors consistently reject malformed requests faster than the simulated attackers can produce them. We thus conclude that Sabre is inherently resistant to DoS attacks in the following sense: *An attacker seeking to overwhelm Sabre servers must expend strictly more resources than the servers.*

#### 7.6.4 Head-to-head with Riposte and Express

The next set of experiments provide a head-to-head comparison between Sabre with 2-verifier SNIPs and both Riposte and Express, the respective state-of-the-art systems in the bulletin board and mailbox models. We plot the results in Figures 7.7 and 7.8. (Note that we measured the reference implementations of Riposte<sup>4</sup> and Express<sup>5</sup> on the same servers we used to run Sabre. The  $x$ -axes on all plots stop at  $n = 2^{18}$  due to limitations of the Riposte and Express implementations.)

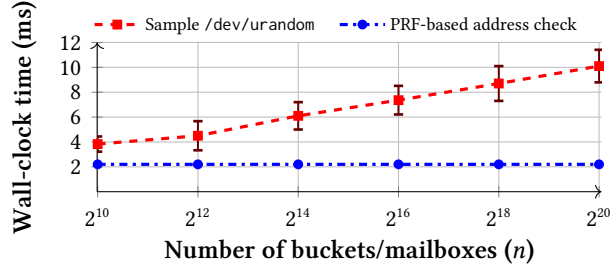
Figure 7.7a compares the throughput of Sabre-M against that of Express for messages of size 1 KiB and 32 KiB. In both cases, Sabre-M comfortably outperforms

---

<sup>4</sup><https://bitbucket.org/henrycg/riposte/>

<sup>5</sup><https://github.com/SabaEskandarian/Express>





(a) PRF-based address check versus malformed request sampling time

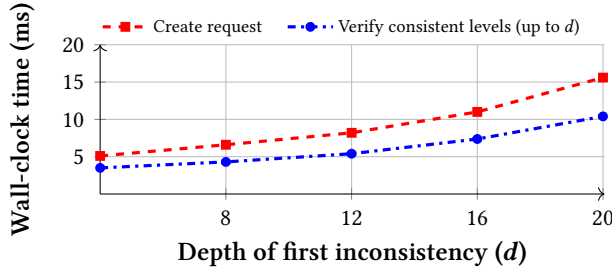
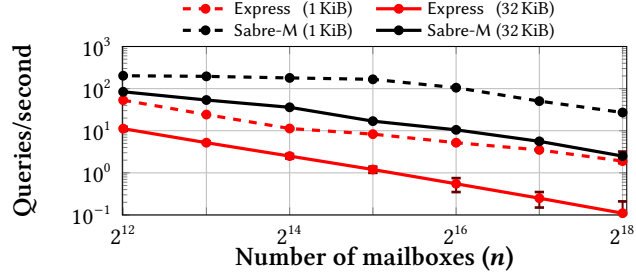
(b) DPF auditing versus malformed request sampling time ( $n = 2^{20}$ )

Figure 7.6: Sabre 2-verifier SNIP auditing versus malformed request sampling. 7.6a compares the cost of checking a batch of 128 mailbox addresses with that of sampling a (single) malformed request; 7.6b compares the cost of verifying a proof up to level  $d$  with that of sampling a malformed proof that verifies up to level  $d$ .

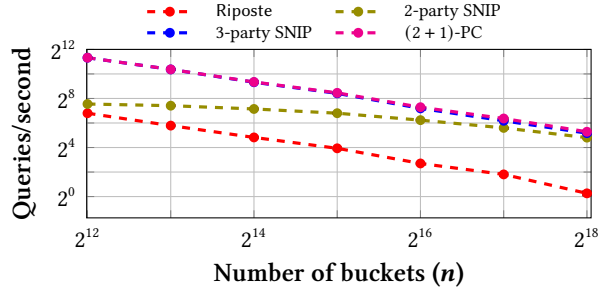
Express; indeed, Sabre-M with 32 KiB messages even outperforms Express with 1 KiB messages.

Figure 7.7b compares the throughput of Sabre-BB against that of Riposte. For a meaningful comparison, we provide separate plots for Sabre-BB with each of the three audit protocol instantiations. Again, the results confirm our expectations: Throughput for the  $(2 + 1)$ -party and 3-verifier SNIP variants consistently exceeds  $20\times$  that of Riposte, while the overhead of cut-and-choose in the 2-verifier SNIP variant eliminates much of this gain when the number of buckets is small. As the number of buckets grows large, the costs of all three Sabre instances converge to that of the full-domain evaluation of the DPF and subsequent writing.

Figure E.1 in Appendix E compares the performance of the “stepping-stones” to Sabre-M. Comparing those plots with the plot for Express (from Figure 7.7a) with



(a) Sabre-M vs. Express (1 KiB &amp; 32 KiB messages)



(b) Sabre-BB vs. Riposte (256 byte messages)

Figure 7.7: Comparison among Sabre variants, Riposte, and Express when all requests are valid.

1 KiB messages reveals comparable throughput when all DPFs pass auditing. This is unsurprising, as the bottleneck in both protocols is the DPF evaluation and subsequent writing. We stress that our use of bitsliced LowMC is crucial here: Express benefits from the fast AES-NI instruction set on modern x86-64 CPUs; without bitslicing, the throughput of LowMC could not compete. The algorithmic improvements of Sabre become apparent as we progress through the “stepping-stones” toward Sabre-M, whose smaller DPFs and ability to use full-domain evaluation greatly reduces the cost of DPF evaluation; moreover, the use of full-domain evaluation provides superior pipe-lining and cache utilization, resulting in notable speedups for the subsequent writing. Our next experiments reveal further algorithmic improvements that arise only when some DPFs fail auditing.



Figure 7.8: Varying the percentage of malformed queries for DPFs of size  $2^{16}$  and message size of 10 KiB. Notice the log-scaling on the  $y$ -axis.

### Head-to-head under DoS attacks

The final set of experiments compares the throughput of Sabre-M with that of Express in the presence of a resource-exhaustion DoS attack. Recall that write requests in the mailbox model can be malformed in two distinct ways; namely, (i) the mailbox address is incorrect or (ii) the DPF is not well-formed. Figure 7.8 presents five plots that each show the effect of varying the proportion of incoming requests that are well-formed versus malformed. In the leftmost plot, all bad requests have valid addresses but malformed DPFs; in the next plot, one in four have invalid addresses; and so on until the rightmost plot where all bad requests have invalid addresses. The  $y$ -axis tracks how many incoming queries (whether “good” or “bad”) each protocol was able to process per second.

Notice that, in all cases, throughput actually *increases* with the proportion of malformed DPFs, since the servers can stop processing immediately if auditing fails. (The effect is more pronounced for Sabre because of its faster auditing.) When the proportion of requests with invalid addresses increases, the throughput for Express plummets, while the throughput for Sabre-M skyrockets. This results from the

extremely low cost of the PRF-based address check that Sabre-M employs. These findings confirm that Sabre-M performs significantly better than Express in the presence of a DoS attack.

## 7.7 Conclusion

We presented Sabre, a family of SAM protocols with instances operating in both the sender-anonymous mailbox model (Sabre-M) and the sender-anonymous bulletin board model (Sabre-BB). Sabre improves on the state-of-the-art systems in both models via several innovations that improve not only its concrete performance and scalability under “ideal” circumstances but also its resilience to resource exhaustion-style DoS attacks in the sender-anonymous mailbox model. Our implementation and experimental analysis indicate that Sabre can feasibly scale to anonymity sets in the tens of millions and beyond.

## Chapter 8

# Conclusion

This thesis examines the intersection of MPC and DPFs in the context of PETs. This has been focused on using DPFs in MPC protocols to enable fast computation of non-linear functions and efficient secret share conversion and on using MPC to generate DPF key pairs and assert the correctness of untrusted DPF key pairs. The overarching goal of this work is to demonstrate to readers how MPC protocols interacting with the internal structure of Boyle–Gilboa–Ishai DPFs can achieve significant performance improvements for components which are critical in a wide variety of PETs. Notable examples include DPF key generating for sender-anonymous messaging protocols and using MPC to compute activation functions for privacy preserving machine learning systems.

### 8.1 GROTTO related work

GROTTO is the latest in a line of works that [11, 35, 41, 56, 60, 62] that use *piecewise-polynomial functions* (or *splines*) for approximating non-linear univariate functions on input additively shared secrets.

Prior works in this area have generally taken one of two possible approaches. The first strategy, introduced by Vadapalli, Bayatbabolghani, and Henry [60] for their Pirsona scheme, uses linear-sized DPFs with an exponential-cost *full-domain evaluation* procedure to evaluate reciprocal square roots and integer comparison. The follow-up scheme Pika [62] generalizes Pirsona’s approach to arbitrary functions

and also adds machinery to thwart malicious dealers. This DPF-plus-full-domain-evaluation approach has low communication cost and concretely efficient running times for “short” inputs (say, 16–24 bits), but its exponential computation cost quickly grows untenable as inputs get longer; indeed, Wagh writes that “typical sizes for which [this approach] provides performance comparable to general purpose (sic) MPC are around 20–25 bits” [62; §3]. The recently proposed ORCA scheme of Jawalkar, Gupta, Basu, Chandran, Gupta, and Sharma [41] uses massive parallelism afforded by GPUs to partially reign in the exponential cost of full-domain evaluation.

The second strategy, exemplified by Gupta, Kumaraswamy, Gupta, and Chandran’s LLAMA scheme [35], uses *distributed comparison functions* (DCFs)—a DPF-adjacent primitive for efficient integer comparison—to avoid the need for costly full-domain evaluations. Swapping out DPFs in favour of DCFs dramatically improves computational scaling at the expense of substantially higher communication costs (that increase rapidly as approximations become more granular).

Compared with its progenitors, GROTTTO’s parity-segment approach offers concretely lower costs for “simple” functions and superior asymptotics as inputs get longer and approximations more elaborate. The upshot is that, while the respectively high computation and communication costs of the Pirsona and LLAMA frameworks severely curtail those schemes’ practically achievable approximation accuracy, the fidelity of GROTTTO approximations is practically limited only by the difficulty of building good piecewise-polynomial approximations—indeed, as seen in Table 5.2, GROTTTO “approximations” are often *errorless* when viewed as fixed-point computations.

## 8.2 Sabre related work

Before concluding, we summarize some relevant works on anonymous messaging. We focus on systems that provide privacy guarantees comparable to those of Sabre, as opposed to schemes like Vuvuzela [61], Stadium [59], or Karaoke [47], which tackle similar problems while providing only differential privacy-like anonymity guarantees.

As described in Chapter 7, the two primary works that relate to Sabre are Riposte and Express. Both of these systems are based upon using two or more servers, each

of which hold a secret shared database of messages. Writes are then performed using DPFs. In Express the DPF’s distinguished point corresponds to the address of the mailbox address being written to.

There are also several related works that take different approaches to SAM protocols. These systems are based upon different assumptions and cryptographic primitives.

*Pung*: Angel and Setty present Pung [3], a single-server messaging protocol built from computationally private information retrieval (CPIR). Pung is notable for its low writing costs (though its reliance on CPIR renders reading expensive) and absence of multiple servers, and resulting non-collusion assumptions.

*Atom*: Kwon, Corrigan-Gibbs, Devadas, and Ford present Atom [45], an anonymous microblogging service in the sender-anonymous bulletin board model. Atom is notable for its use of sharding and public-key techniques to support anonymous broadcasting of “short” (Tweet-sized) messages in the presence of actively malicious servers.

*XRD*: Kwon, Lu, and Devadas present XRD [46], a scalable end-to-end messaging system. XRD is notable for its use of parallel mixes and so-called aggregate hybrid shuffles to achieve high throughput relative to Pung and Atom.

*Talek*: Cheng, Scott, Masserova, Zhang, Goyal, Anderson, Krishnamurthy, and Parno present Talek [17], a group messaging system in the anonymous mailbox model. Talek is notable for its use of information-theoretic PIR techniques to achieve access sequence indistinguishability under weak non-collusion assumptions.

Compared to prior SAM protocols, Sabre achieves significantly better performance than prior works, as we demonstrate in [Section 7.6](#). In particular, when targeted by a DoS attack, Sabre’s auditing protocol significantly outperforms the auditing protocols from Riposte and Express.

### 8.3 Future Work

There are several paths for future works in this area to explore. Some of these areas expand upon Sabre and others expand upon GROTT.

**Combining Sabre with other auditing protocols.** In [Chapter 7](#), we demonstrate how Sabre’s auditing protocol significantly outperforms prior auditing protocols. This is especially notable in the case of a DoS attack that sends a large number of invalid write requests. In Express, a large part of this difference is that Express’s audit protocol requires evaluating the DPF on all registered mailbox addresses. When the DPF is used for a valid write after the audit, the results from the evaluation step can be reused for the write. This makes overall cost of the entire writing process more efficient for Express when there are extremely few invalid write requests, but it makes Express much less efficient than Sabre when the percentage of invalid requests rises. Combining these two techniques into one system could allow for dynamic switching between the two auditing methods in order to optimize the writing process depending on the proportion of invalid requests being received.

**Applications of GROTTO.** There are a variety of applications for GROTTO in MPC. One application of particular interest is privacy-preserving machine learning based upon MPC. GROTTO is highly applicable because neural network-based machine learning makes heavy use of non-linear activation functions, which, as shown in [Section 5.4](#), GROTTO can evaluate extremely efficiently.

**Extension of GROTTO to periodic functions.** [Chapter 6](#) shows how GROTTO can be used to enable efficient bit extraction and bit decomposition. The techniques used to make bit decomposition efficient rely upon the fact that bit extraction is a periodic function whose period is a power of two. This makes it easy to reduce the bitlength of inputs to the GROTTO bit extraction function in order to improve efficiency. If this technique can be generalized to other periodic functions, where the period is not a power of two, it would vastly improve upon existing techniques for evaluating periodic functions.

**Enable malicious security in GROTTO.** As described in [Section 5.0.2](#), GROTTO’s threat model assumes semi-honest adversaries. However, through the use of techniques based upon *linear sketching* [[14](#), [21](#), [62](#)] or *secret-shared non-interactive proofs* [[25](#)], this assumption can be relaxed to allow for security against malicious actively adversaries. The DPF verification protocol from



[Chapter 7](#) is another example of the available methods based upon secret-shared non-interactive proofs.

## 8.4 Conclusion

This work has shown how DPFs and MPC can be used in combination to great effect in the Sabre and GROTO systems. The author hopes that the results presented in this thesis will encourage further research into efficient protocols for integrating DPFs into MPC protocols with the goal of making privacy available to everyone through more scalable and efficient PET systems.

# Bibliography

- [1] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. [Ciphers for MPC and FHE](#). In *Advances in Cryptology: Proceedings of EUROCRYPT 2015 (Part I)*, volume 9056 of LNCS, pages 430–454, Sofia, Bulgaria (April 2015).
- [2] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. [MCMix: Anonymous messaging via secure multiparty computation](#). In *Proceedings of USENIX Security 2017*, pages 1217–1234, Vancouver, BC, Canada (August 2017).
- [3] Sebastian Angel and Srinath T. V. Setty. [Unobservable communication over fully untrusted infrastructure](#). In *Proceedings of OSDI 2016*, pages 551–569, Savannah, GA, USA (November 2016).
- [4] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. [More efficient oblivious transfer and extensions for faster secure computation](#). In *Proceedings of CCS 2013*, pages 535–548, 2013.
- [5] Mikhail Atallah, Marina Bykova, Jiangtao Li, Keith Frikken, and Mercan Topkara. Private collaborative forecasting and benchmarking. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES '04*, page 103–114, 2004. Association for Computing Machinery.
- [6] Donald Beaver. [Efficient multiparty protocols using circuit randomization](#). In *Advances in Cryptology: Proceedings of CRYPTO 1991*, volume 576 of LNCS, pages 420–432, Santa Barbara, CA, USA (August 1991).

- 
- [7] Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 479–488, 1996.
  - [8] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 503–513, 1990.
  - [9] George Robert Blakley. Safeguarding cryptographic keys. In *Managing Requirements Knowledge, International Workshop on*, pages 313–313. IEEE Computer Society, 1979.
  - [10] Sergey Bochkanov and the ALGLIB Project development team. ALGLIB; version 3.19.0 [computer software]. Available from: <https://www.alglib.net/>, June 2022.
  - [11] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. [Function secret sharing for mixed-mode and fixed-point secure computation](#). In *Advances in Cryptology: Proceedings of EUROCRYPT 2021 (Part II)*, volume 12697 of LNCS, pages 871–900, Zagreb, Croatia (October 2021).
  - [12] Elette Boyle, Niv Gilboa, and Yuval Ishai. [Function secret sharing](#). In *Advances in Cryptology: Proceedings of EUROCRYPT 2015 (Part II)*, volume 9057 of LNCS, pages 337–367, Sofia, Bulgaria (April 2015).
  - [13] Elette Boyle, Niv Gilboa, and Yuval Ishai. [Breaking the circuit size barrier for secure computation under DDH](#). In *Advances in Cryptology: Proceedings of CRYPTO 2016 (Part I)*, volume 9814 of LNCS, pages 509–539, Santa Barbara, CA, USA (August 2016).
  - [14] Elette Boyle, Niv Gilboa, and Yuval Ishai. [Function secret sharing: Improvements and extensions](#). In *Proceedings of CCS 2016*, pages 1292–1303, Vienna, Austria (October 2016).
  - [15] Julien Bringer, Herve Chabanne, Melanie Favre, Alain Patey, Thomas Schneider, and Michael Zohner. Gshade: Faster privacy-preserving distance computation and biometric identification. In *Proceedings of the 2nd ACM Workshop on Information Hiding and Multimedia Security, IH&MMSec ’14*, page 187–198, 2014. Association for Computing Machinery.

- [16] Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, and Greg Zaverucha. [Post-quantum zero-knowledge and signatures from symmetric-key primitives](#). In *Proceedings of CCS 2017*, pages 1825–1842, Dallas, TX, USA (October 2017).
- [17] Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas E. Anderson, Arvind Krishnamurthy, and Bryan Parno. [Talek: Private group messaging with hidden access patterns](#). In *Proceedings of ACSAC 2020*, pages 84–99, Austin, TX, USA (December 2020).
- [18] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. [Private information retrieval](#). In *Proceedings of FOCS 1995*, pages 41–50, Milwaukee, WI, USA (October 1995).
- [19] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. [Private information retrieval](#). *Journal of the ACM (JACM)*, 45(6):965–981, November 1998.
- [20] Henry Corrigan-Gibbs and Dan Boneh. [Prio: Private, robust, and scalable computation of aggregate statistics](#). In *Proceedings of USENIX Security 2017*, pages 259–282, Boston, MA, USA (March 2017).
- [21] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. [Riposte: An anonymous messaging system handling millions of users](#). In *Proceedings of IEEE S&P 2015*, pages 321–338, San Jose, CA, USA (May 2015).
- [22] Daniel Demmler, Thomas Schneider, and Michael Zohner. [ABY —A framework for efficient mixed-protocol secure two-party computation](#). In *Proceedings of NDSS 2015*, San Diego, CA, USA (February 2015).
- [23] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. [Tor: The second-generation onion router](#). In *Proceedings of USENIX Security 2004*, San Diego, CA, USA (August 2004).
- [24] Wenliang Du and Mikhail J. Atallah. [Protocols for secure remote database access with approximate matching](#). In *E-Commerce Security and Privacy (Part II)*, volume 2 of *Advances in Information Security*, pages 87–111, February 2001.

- [25] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. [Express: Lowering the cost of metadata-hiding communication with cryptographic privacy](#). In *Proceedings of USENIX Security 2021*, Vancouver, BC, Canada (August 2021).
- [26] Amos Fiat and Adi Shamir. [How to prove yourself: Practical solutions to identification and signature problems](#). In *Advances in Cryptology: Proceedings of CRYPTO 1986*, volume 263 of *LNCS*, pages 186–194, Santa Barbara, CA, USA (August 1986).
- [27] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. [ZKBoo: Faster zero-knowledge for boolean circuits](#). In *Proceedings of USENIX Security 2016*, pages 1069–1083, Austin, TX, USA (August 2016).
- [28] Niv Gilboa. Two party rsa key generation. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO’ 99*, pages 116–129, 1999. Springer Berlin Heidelberg.
- [29] Niv Gilboa and Yuval Ishai. [Distributed point functions and their applications](#). In *Advances in Cryptology: Proceedings of EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658, Copenhagen, Denmark (May 2014).
- [30] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR)*, 23(1):5–48, 1991.
- [31] Oded Goldreich. *The Foundations of Cryptography – Volume 1, Basic Techniques*. Cambridge University Press, New York, NY, USA, June 2001.
- [32] Oded Goldreich. *The Foundations of Cryptography – Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, May 2004.
- [33] Oded Goldreich, Silvio Micali, and Avi Wigderson. [How to play any mental game or a completeness theorem for protocols with honest majority](#). In *Proceedings of STOC 1987*, pages 218–229, New York, NY, USA (May 1987).
- [34] Torbjörn Granlund and the GMP development team. GNU MP: The GNU Multiple Precision Arithmetic Library; version 4.6.1 [computer software]. Available from: <https://gmplib.org/>, November 2020.

- 
- [35] Kanav Gupta, Deepak Kumaraswamy, Divya Gupta, and Nishanth Chandran. [LLAMA: A low latency math library for secure inference](#). *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2022(4):274–294, October 2022.
- [36] Ryan Henry and Adithya Vadapalli. dpf++; version 0.0.1 [computer software]. Available from: <https://www.github.com/rh3nry/dpfplusplus>, July 2019.
- [37] William George Horner. [A new method of solving numerical equations of all orders, by continuous approximation](#). *Philosophical Transactions of the Royal Society of London*, 109:308–335, January 1819.
- [38] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. [Extending oblivious transfers efficiently](#). In *Advances in Cryptology: Proceedings of CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161, Santa Barbara, CA, USA (August 2003).
- [39] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. [Zero-knowledge from secure multiparty computation](#). In *Proceedings of STOC 2007*, pages 21–30, San Diego, CA, USA (June 2007).
- [40] Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.
- [41] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. [Orca: FSS-based Secure Training with GPUs](#). *IACR Cryptology ePrint Archive*, Report 2023/206, February 2023.
- [42] Jonathan Katz, Vladimir Kolesnikov, and Xiao Wang. [Improved non-interactive zero knowledge with applications to post-quantum signatures](#). In *Proceedings of CCS 2018*, pages 525–537, Toronto, ON, Canada (October 2018).
- [43] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, September 2014.
- [44] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35*, pages 486–498. Springer, 2008.

- 
- [45] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. [Atom: Horizontally scaling strong anonymity](#). In *Proceedings of SOSP 2017*, pages 406–422, Shanghai, China (October 2017).
  - [46] Albert Kwon, David Lu, and Srinivas Devadas. [XRD: Scalable messaging system with cryptographic privacy](#). In *Proceedings of NSDI 2020*, pages 759–776, Santa Clara, CA, USA (February 2020).
  - [47] David Lazar, Yossi Gilad, and Nickolai Zeldovich. [Karaoke: Distributed private messaging immune to passive traffic analysis](#). In *Proceedings of OSDI 2018*, pages 711–725, Carlsbad, CA, USA (October 2018).
  - [48] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. *Journal of Cryptology*, 28(2):312–350, 2015.
  - [49] Stephen Matyas, Carl Meyer, and Jonathan Oseas. Generating strong one-way functions with cryptographic algorithm. *IBM Technical Disclosure Bulletin*, 27(10A):5658–565, March 1985.
  - [50] Payman Mohassel and Peter Rindal.  $\text{Aby}^3$ : A mixed protocol framework for machine learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, page 35–52, 2018. Association for Computing Machinery.
  - [51] Payman Mohassel and Yupeng Zhang. [SecureML: A system for scalable privacy-preserving machine learning](#). In *Proceedings of IEEE S&P 2017*, pages 19–38, San Jose, CA, USA (May 2017).
  - [52] Pascal Paillier. [Public-key cryptosystems based on composite degree residuosity classes](#). In *Advances in Cryptology: Proceedings of EUROCRYPT 1999*, volume 1592 of LNCS, pages 223–238, Prague, Czech Republic (May 1999).
  - [53] Arpita Patra and Ajith Suresh. [BLAZE: Blazing fast privacy-preserving machine learning](#). In *Proceedings of NDSS 2020*, San Diego, CA, USA (February 2020).
  - [54] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. [ABY2.0: Improved mixed-protocol secure two-party computation](#). In *Proceedings of USENIX Security 2021*, pages 2165–2182, Virtual event (August 2021).

- [55] Michael O. Rabin. [How to exchange secrets with oblivious transfer](#). Technical Report TR-81, Aiken Computation Lab, Harvard University, Cambridge, MA, USA, May 1981.
- [56] Théo Ryffel, Pierre Tholoniati, David Pointcheval, and Francis R. Bach. [AriaNN: Low-interaction privacy-preserving deep learning via function secret sharing](#). *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2022(1):291–316, January 2022.
- [57] Adi Shamir. [How to share a secret](#). *Communications of the ACM (CACM)*, 22(11):612–613, November 1979.
- [58] Neil J. A. Sloane and OEIS Foundation Inc. [A083652: Sum of lengths of binary expansions of 0 through  \$n\$](#) . In *The on-line encyclopedia of integer sequences*. [Online; accessed 2022-10-13].
- [59] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. [Stadium: A distributed metadata-private messaging system](#). In *Proceedings of SOSP 2017*, pages 423–440, Shanghai, China (October 2017).
- [60] Adithya Vadapalli, Fattaneh Bayatbabolghani, and Ryan Henry. [You may also like... Privacy: Recommendation systems meet PIR](#). *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2021(4):30–53, October 2021.
- [61] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. [Vuvuzela: Scalable private messaging resistant to traffic analysis](#). In *Proceedings of SOSP 2015*, Monterey, CA, USA (October 2015).
- [62] Sameer Wagh. [Pika: Secure computation using function secret sharing over rings](#). *Proceedings on Privacy Enhancing Technologies (PoPETS)*, 2022(4):351–377, October 2022.
- [63] Andrew Chi-Chih Yao. [Protocols for secure computations \(Extended abstract\)](#). In *Proceedings of FOCS 1982*, pages 160–164, Chicago, IL, USA (November 1982).
- [64] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th annual symposium on foundations of computer science (Sfcs 1986)*, pages 162–167. IEEE, 1986.



- [65] Randy Yates. Fixed-point arithmetic: An introduction. *Digital Signal Labs*, 81(83):198, 2009.
- [66] Samee Zahur, Mike Rosulek, and David Evans. [Two halves make a whole: Reducing data transfer in garbled circuits using half gates](#). In *Advances in Cryptology: Proceedings of EUROCRYPT 2015 (Part II)*, volume 9057 of *LNCS*, pages 220–250, Sofia, Bulgaria (April 2015).

## Appendix A

# Prefix-parity amortization via memoization

This appendix presents empirically measured total costs for the prefix-parity algorithm with many endpoints, such as when evaluating several complex functions at once. The costs depend heavily on the distribution of the points, with higher endpoint density implying greater per-endpoint savings. We choose three kinds of distributions, namely uniform, Gaussian, and Zipfian.

As expected, the greater the variance of the distribution, the lower the amortization savings that the prefix-parity algorithm enjoys, with the “worst” case occurring when endpoints are sampled uniformly. By contrast, the amortization savings for Zipfian distributions with small parameters are very extreme.

## A. Prefix-parity amortization via memoization

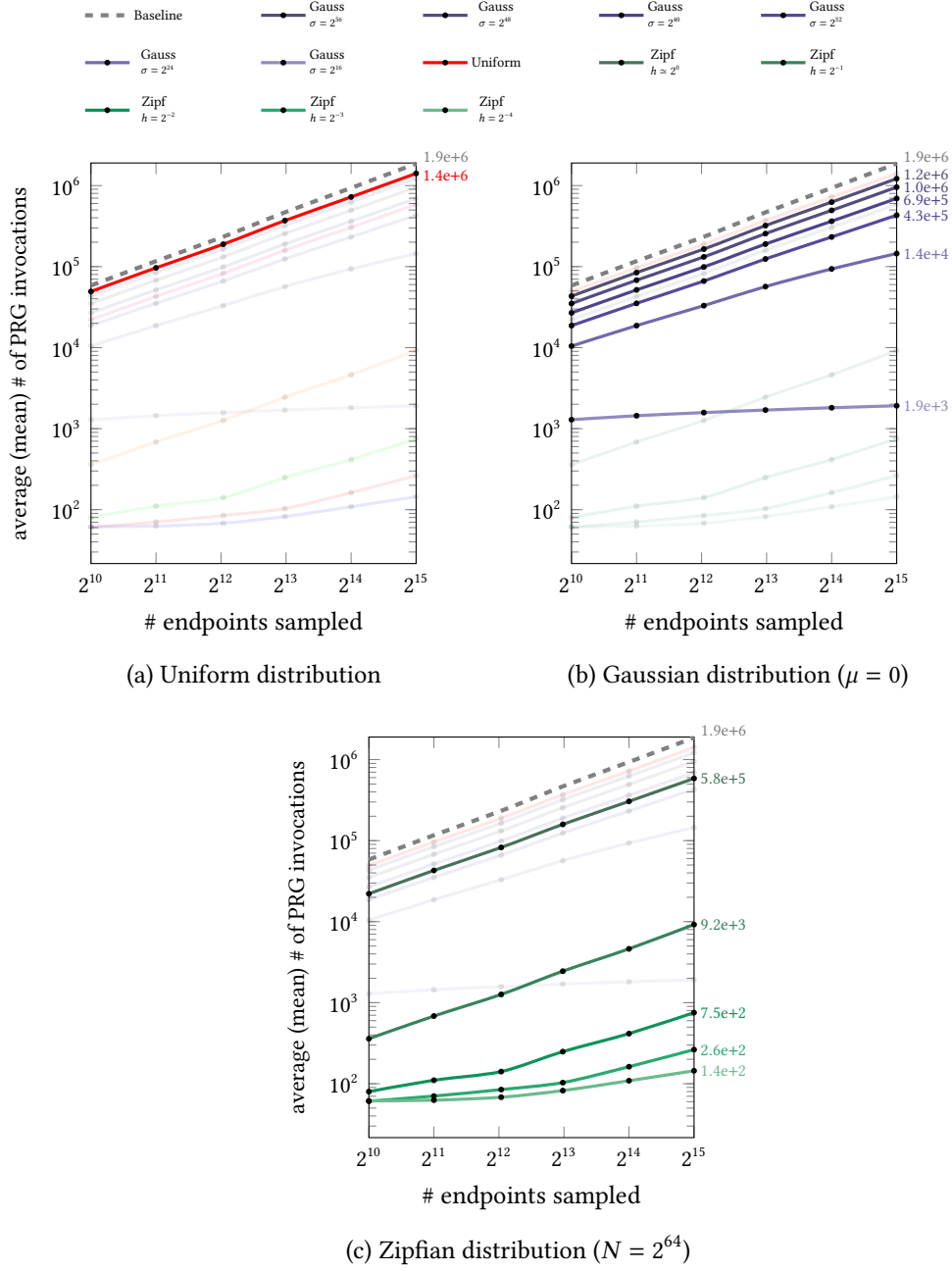


Figure A.1: Empirical amortized costs for prefix-parity over points sampled from selected probability distributions.

## Appendix B

# Formulae for (2+1)-PC sign-corrected polynomial evaluation

Table B.1: Comparing round complexity and communication cost to evaluate polynomials using  $(2 + 1)$ -party protocols. The communication cost is expressed as the number of values exchanged as part of the Beaver triple-like from  $P_2$  (in the preprocessing phase) and between  $P_0$  and  $P_1$  (in the online phase). The bitlength of each term depends on the polynomial degree and fractional precision; see [Section 5.3.1](#) for details on how to calculate them.

Protocol	Polynomial degree	Round complexity	Communication	
			Preprocessing	Online
One-round ABY2.0-like	1	1	8	4
	2	1	13	5
	3	1	18	6
Two-round ABY2.0-like	1	2	6	4
	2	2	9	5
	3	2	13	6
Horner's method	1	2	6	4
	2	3	8	5
	3	4	10	6

## B.1 One-round ABY2.0-like evaluation

### B.1.1 For constant polynomials

**Precomputation:**  $P_2$  samples  $U$  and  $A_0$ , computes  $[u]$ ,  $[A_0]$ ,  $[u \cdot A_0]$ , and shares all five them among  $P_0$  and  $P_1$ .

**Round 1:**  $P_b$  sends  $[\bar{u}]_b = [u]_b + [U]_b$  and  $[\bar{a}_0]_b = [a_0]_b + [A_0]_b$  to  $P_{1-b}$ , and vice versa.

**Output:**  $P_b$  outputs  $[y]_b = \bar{u} \cdot \bar{a}_0 \cdot b - \bar{u} \cdot [A_0]_0 - \bar{a}_0 \cdot [u]_0 + [u \cdot A_0]_0$

The result is  $[y] = \bar{u} \cdot \bar{a}_0 - \bar{u} \cdot [A_0] - \bar{a}_0 \cdot [u] + [u \cdot A_0]$

This is derived by:

$$\begin{aligned} [y] &= [y]_0 + [y]_1 \\ &= \bar{u} \cdot \bar{a}_0 - \bar{u} \cdot [A_0] - \bar{a}_0 \cdot [u] + [u \cdot A_0] \\ &= (\bar{u} - [u]) \cdot (\bar{a}_0 - [A_0]) \\ &= u \cdot a_0 \end{aligned}$$

### B.1.2 For linear polynomials

**Precomputation:**  $P_2$  samples  $A_0, A_1, U$ , and  $X$ , computes  $[u \cdot X]$ ,  $[u \cdot A_1]$ ,  $[A_1 \cdot X - A_0]$ ,  $[U \cdot (A_1 \cdot X - A_0)]$ , and shares all eight among  $P_0$  and  $P_1$ .

**Round 1:**  $P_b$  sends  $[\bar{u}]_b = [u]_b + [U]_b$ ,  $[\bar{a}_0]_b = [a_0]_b + [A_0]_b$ ,  $[\bar{a}_1]_b = [a_1]_b + [A_1]_b$ , and  $[\bar{x}]_b = [x]_b + [X]_b$  to  $P_{1-b}$ , and vice versa.

**Output:**  $P_b$  outputs  $[y]_b = \bar{u} \cdot (b \cdot (\bar{a}_1 \cdot \bar{x} + \bar{a}_0) - \bar{a}_1 \cdot [X]_b - \bar{x} \cdot [A_1]_b + [A_1 \cdot X - A_0]_b) - (\bar{a}_1 \cdot \bar{x} + \bar{a}_0) \cdot [u]_b + \bar{a}_1 \cdot [u \cdot X]_b + \bar{x} \cdot [u \cdot A_1]_b - [U \cdot (A_1 \cdot X - A_0)]_b$

The result is  $[y] = \bar{u} \cdot ((\bar{a}_1 \cdot \bar{x} + \bar{a}_0) - \bar{a}_1 \cdot [X] - \bar{x} \cdot [A_1] + [A_1 \cdot X - A_0]) - (\bar{a}_1 \cdot \bar{x} + \bar{a}_0) \cdot [u] + \bar{a}_1 \cdot [u \cdot X] + \bar{x} \cdot [u \cdot A_1] - [U \cdot (A_1 \cdot X - A_0)]$

This is derived by:

$$\begin{aligned}
 [y] &= [y]_0 + [y]_1 \\
 &= \bar{u} \cdot (\bar{a}_1 \cdot \bar{x} + \bar{a}_0 - \bar{a}_1 \cdot [X] - \bar{x} \cdot [A_1] + [A_1 \cdot X - A_0]) \\
 &\quad - (\bar{a}_1 \cdot \bar{x} + \bar{a}_0) \cdot [u] + \bar{a}_1 \cdot [u \cdot X] + \bar{x} \cdot [u \cdot A_1] \\
 &\quad - [U \cdot (A_1 \cdot X - A_0)] \\
 &= \bar{u} \cdot \bar{a}_1 \cdot \bar{x} - \bar{u} \cdot \bar{a}_1 \cdot [X] - \bar{u} \cdot \bar{x} \cdot [A_1] - \bar{a}_1 \cdot \bar{x} \cdot [u] \\
 &\quad + \bar{u} \cdot [A_1 \cdot X] + \bar{a}_1 \cdot [u \cdot X] + \bar{x} \cdot [u \cdot A_1] - [u \cdot A_1 \cdot X] \\
 &\quad + \bar{u} \cdot \bar{a}_0 - \bar{u} \cdot [A_0] - \bar{a}_0 \cdot [u] + [u \cdot A_0] \\
 &= \bar{u} \cdot \bar{a}_1 \cdot \bar{x} - \bar{u} \cdot \bar{a}_1 \cdot [X] - \bar{u} \cdot \bar{x} \cdot [A_1] - \bar{a}_1 \cdot \bar{x} \cdot [u] \\
 &\quad + \bar{u} \cdot [A_1 \cdot X] + \bar{a}_1 \cdot [u \cdot X] + \bar{x} \cdot [u \cdot A_1] - [u \cdot A_1 \cdot X] \\
 &\quad + u \cdot a_0 \cdot 2^p \\
 &= (\bar{u} - [u]) \cdot (\bar{a}_1 - [A_1]) \cdot (\bar{x} - [X]) + u \cdot a_0 \cdot 2^p \\
 &= u \cdot a_1 \cdot x + u \cdot a_0 \cdot 2^p \\
 &= u \cdot (a_1 \cdot x + a_0 \cdot 2^p)
 \end{aligned}$$

### B.1.3 For quadratic polynomials

**Precomputation:**  $P_2$  samples  $A_0, A_1, A_2, U$ , and  $X$ , computes  $[X^2], [u \cdot A_2], [u \cdot X], [U \cdot X^2], [2 \cdot A_2 \cdot X - A_1], [A_2 \cdot X^2 - A_1 \cdot X + A_0], [u \cdot (2 \cdot A_2 \cdot X - A_1)], [u \cdot (A_2 \cdot X^2 - A_1 \cdot X + A_0)]$ , and shares all thirteen among  $P_0$  and  $P_1$ .

**Round 1:**  $P_b$  sends  $[\bar{u}]_b = [u]_b + [U]_b, [\bar{a}_0]_b = [a_0]_b + [A_0]_b, [\bar{a}_1]_b = [a_1]_b + [A_1]_b, [\bar{a}_2]_b = [a_2]_b + [A_2]_b$ , and  $[\bar{x}]_b = [x]_b + [X]_b$  to  $P_{1-b}$ , and vice versa.

**Output:**  $P_b$  outputs  $[y]_b = \bar{u} \cdot (b \cdot (\bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) - \bar{x}^2 \cdot [A_2]_b + \bar{x} \cdot [2 \cdot A_2 \cdot X - A_1]_b - (\bar{a}_1 + 2 \cdot \bar{a}_2 \cdot \bar{x}) \cdot [X]_b + \bar{a}_2 \cdot [X^2]_b - [A_2 \cdot X^2 - A_1 \cdot X + A_0]_b) - [u]_b \cdot (\bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) + \bar{x}^2 \cdot [u \cdot A_2]_b - \bar{x} \cdot [u \cdot (2 \cdot A_2 \cdot X - A_1)]_b + (\bar{a}_1 + 2 \cdot \bar{a}_2 \cdot \bar{x}) \cdot [u \cdot X]_b - \bar{a}_2 \cdot [U \cdot X^2]_b + [u \cdot (A_2 \cdot X^2 - A_1 \cdot X + A_0)]_b$

The result is  $[y] = \bar{u} \cdot ((\bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) - \bar{x}^2 \cdot [A_2] + \bar{x} \cdot [2 \cdot A_2 \cdot X - A_1] - (\bar{a}_1 + 2 \cdot \bar{a}_2 \cdot \bar{x}) \cdot [X] + \bar{a}_2 \cdot [X^2] - [A_2 \cdot X^2 - A_1 \cdot X + A_0]) - [u] \cdot (\bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) + \bar{x}^2 \cdot [u \cdot A_2] - \bar{x} \cdot [u \cdot (2 \cdot A_2 \cdot X - A_1)] + (\bar{a}_1 + 2 \cdot \bar{a}_2 \cdot \bar{x}) \cdot [u \cdot X] - \bar{a}_2 \cdot [U \cdot X^2] + [u \cdot (A_2 \cdot X^2 - A_1 \cdot X + A_0)]$

This is derived by:

$$\begin{aligned}
 [y] &= [y]_0 + [y]_1 \\
 &= \bar{u} \cdot ((\bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) - (2 \cdot \bar{x} \cdot \bar{a}_2 + \bar{a}_1) \cdot [X] \\
 &\quad + \bar{a}_2 \cdot [X^2] - \bar{x}^2 \cdot [A_2] + \bar{x} \cdot [2 \cdot A_2 \cdot X - A_1] \\
 &\quad - [A_2 \cdot X^2 - A_1 \cdot X + A_0]) \\
 &\quad - (\bar{a}_2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) \cdot \bar{x}^2 [u] \\
 &\quad + (2 \cdot \bar{x} \cdot \bar{a}_2 + \bar{a}_1) \cdot [u \cdot X] - \bar{a}_2 \cdot [U \cdot X^2] \\
 &\quad + \bar{x}^2 \cdot [u \cdot A_2] - \bar{x} \cdot [u \cdot (2 \cdot A_2 \cdot X - A_1)] \\
 &\quad + [U \cdot (A_2 \cdot X^2 - A_1 \cdot X + A_0)] \\
 &= \bar{u} \cdot (\bar{a}_2 \cdot \bar{x}^2 - 2 \cdot \bar{x} \cdot \bar{a}_2 \cdot [X] + \bar{a}_2 \cdot [X^2] \\
 &\quad - \bar{x}^2 \cdot [A_2] + 2 \cdot \bar{x} \cdot [A_2 \cdot X] - [A_2 \cdot X^2]) \\
 &\quad - \bar{a}_2 \cdot \bar{x}^2 [u] + 2 \cdot \bar{x} \cdot \bar{a}_2 \cdot [u \cdot X] \\
 &\quad - \bar{a}_2 \cdot [U \cdot X^2] + \bar{x}^2 \cdot [u \cdot A_2] - 2 \cdot \bar{x} \cdot [u \cdot A_2 \cdot X] \\
 &\quad + [U \cdot A_2 \cdot X^2] + \bar{u} \cdot ((\bar{a}_1 \cdot \bar{x} + \bar{a}_0) - \bar{a}_1 \cdot [X] \\
 &\quad - \bar{x} \cdot [A_1] + [A_1 \cdot X - A_0]) - (\bar{a}_1 \cdot \bar{x} + \bar{a}_0) \cdot [u] \\
 &\quad + \bar{a}_1 \cdot [u \cdot X] + \bar{x} \cdot [u \cdot A_1] - [U \cdot (A_1 \cdot X - A_0)] \\
 &= \bar{u} \cdot (\bar{a}_2 \cdot \bar{x}^2 - 2 \cdot \bar{x} \cdot \bar{a}_2 \cdot [X] + \bar{a}_2 \cdot [X^2] \\
 &\quad - \bar{x}^2 \cdot [A_2] + 2 \cdot \bar{x} \cdot [A_2 \cdot X] - [A_2 \cdot X^2]) \\
 &\quad - \bar{a}_2 \cdot \bar{x}^2 [u] + 2 \cdot \bar{x} \cdot \bar{a}_2 \cdot [u \cdot X] - \bar{a}_2 \cdot [U \cdot X^2] \\
 &\quad + \bar{x}^2 \cdot [u \cdot A_2] - 2 \cdot \bar{x} \cdot [u \cdot A_2 \cdot X] + [U \cdot A_2 \cdot X^2] \\
 &\quad + u \cdot (a_1 \cdot x + a_0 \cdot 2^p) \cdot 2^p \\
 &= (\bar{u} \cdot \bar{a}_2 \cdot \bar{x}^2 - 2 \cdot \bar{u} \cdot \bar{x} \cdot \bar{a}_2 \cdot [X] + \bar{u} \cdot \bar{a}_2 \cdot [X^2] \\
 &\quad - \bar{u} \cdot \bar{x}^2 \cdot [A_2] + 2 \cdot \bar{u} \cdot \bar{x} \cdot [A_2 \cdot X] - \bar{u} \cdot [A_2 \cdot X^2]) \\
 &\quad - \bar{a}_2 \cdot \bar{x}^2 [u] + 2 \cdot \bar{x} \cdot \bar{a}_2 \cdot [u \cdot X] - \bar{a}_2 \cdot [U \cdot X^2] \\
 &\quad + \bar{x}^2 \cdot [u \cdot A_2] - 2 \cdot \bar{x} \cdot [u \cdot A_2 \cdot X] + [U \cdot A_2 \cdot X^2] \\
 &\quad + u \cdot (a_1 \cdot x + a_0 \cdot 2^p) \cdot 2^p \\
 &= (\bar{u} - [u]) \cdot (\bar{a}_2 - [A_2]) \cdot (\bar{x} - [X])^2 \\
 &\quad + u \cdot (a_1 \cdot x + a_0 \cdot 2^p) \cdot 2^p
 \end{aligned}$$

$$\begin{aligned}
 &= u \cdot a_2 \cdot x^2 + u \cdot (a_1 \cdot x + a_0 \cdot 2^p) \cdot 2^p \\
 &= u \cdot (a_2 \cdot x^2 + a_1 \cdot x \cdot 2^p + a_0 \cdot 2^{2p})
 \end{aligned}$$

#### B.1.4 For cubic polynomials

**Precomputation:**  $P_2$  samples  $A_0, A_1, A_2, A_3, U$ , and  $X$ , computes  $[X^2], [X^3], [3 \cdot A_3 \cdot X - A_2], [3 \cdot A_3 \cdot X^2 - 2 \cdot A_2 \cdot X + A_1], [A_3 \cdot X^3 - A_2 \cdot X^2 + A_1 \cdot X - A_0], [u \cdot X], [u \cdot X^2], [u \cdot X^3], [u \cdot A_3], [u \cdot (3 \cdot A_3 \cdot X - A_2)], [U \cdot (3 \cdot A_3 \cdot X^2 - 2 \cdot A_2 \cdot X + A_1)], [U \cdot (A_3 \cdot X^3 - A_2 \cdot X^2 + A_1 \cdot X - A_0)]$ , and shares all eighteen among  $P_0$  and  $P_1$ .

**Round 1:**  $P_b$  sends  $[\bar{u}]_b = [u]_b + [U]_b$ ,  $[\bar{a}_0]_b = [a_0]_b + [A_0]_b$ ,  $[\bar{a}_1]_b = [a_1]_b + [A_1]_b$ ,  $[\bar{a}_2]_b = [a_2]_b + [A_2]_b$ ,  $[\bar{a}_3]_b = [a_3]_b + [A_3]_b$ , and  $[\bar{x}]_b = [x]_b + [X]_b$  to  $P_{1-b}$ , and vice versa.

**Output:**  $P_b$  outputs  $[y]_b = \bar{u} \cdot (b \cdot (\bar{a}_3 \cdot \bar{x}^3 + \bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) - (3 \cdot \bar{a}_3 \cdot \bar{x}^2 + 2 \cdot \bar{a}_2 \cdot \bar{x} + \bar{a}_1) \cdot [X]_b + (3 \cdot \bar{a}_3 \cdot \bar{x} + \bar{a}_2) \cdot [X^2]_b - \bar{a}_3 \cdot [X^3]_b - \bar{x}^3 \cdot [A_3]_b + \bar{x}^2 \cdot [3 \cdot A_3 \cdot X - A_2]_b - \bar{x} \cdot [3 \cdot A_3 \cdot X^2 - 2 \cdot A_2 \cdot X + A_1]_b + [A_3 \cdot X^3 - A_2 \cdot X^2 + A_1 \cdot X - A_0]_b) - [u]_b \cdot (\bar{a}_3 \cdot \bar{x}^3 + \bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) + (3 \cdot \bar{a}_3 \cdot \bar{x}^2 + 2 \cdot \bar{a}_2 \cdot \bar{x} + \bar{a}_1) \cdot [u \cdot X]_b - (3 \cdot \bar{a}_3 \cdot \bar{x} + \bar{a}_2) \cdot [U \cdot X^2]_b + \bar{a}_3 \cdot [U \cdot X^3]_b + \bar{x}^3 \cdot [u \cdot A_3]_b - \bar{x}^2 \cdot [u \cdot (3 \cdot A_3 \cdot X - A_2)]_b + \bar{x} \cdot [U \cdot (3 \cdot A_3 \cdot X^2 - 2 \cdot A_2 \cdot X + A_1)]_b - [U \cdot (A_3 \cdot X^3 - A_2 \cdot X^2 + A_1 \cdot X - A_0)]_b$

The result is  $[y] = \bar{u} \cdot ((\bar{a}_3 \cdot \bar{x}^3 + \bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) - (3 \cdot \bar{a}_3 \cdot \bar{x}^2 + 2 \cdot \bar{a}_2 \cdot \bar{x} + \bar{a}_1) \cdot [X] + (3 \cdot \bar{a}_3 \cdot \bar{x} + \bar{a}_2) \cdot [X^2] - \bar{a}_3 \cdot [X^3] - \bar{x}^3 \cdot [A_3] + \bar{x}^2 \cdot [3 \cdot A_3 \cdot X - A_2] - \bar{x} \cdot [3 \cdot A_3 \cdot X^2 - 2 \cdot A_2 \cdot X + A_1] + [A_3 \cdot X^3 - A_2 \cdot X^2 + A_1 \cdot X - A_0]) - [u] \cdot (\bar{a}_3 \cdot \bar{x}^3 + \bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) + (3 \cdot \bar{a}_3 \cdot \bar{x}^2 + 2 \cdot \bar{a}_2 \cdot \bar{x} + \bar{a}_1) \cdot [u \cdot X] - (3 \cdot \bar{a}_3 \cdot \bar{x} + \bar{a}_2) \cdot [U \cdot X^2] + \bar{a}_3 \cdot [U \cdot X^3] + \bar{x}^3 \cdot [u \cdot A_3] - \bar{x}^2 \cdot [u \cdot (3 \cdot A_3 \cdot X - A_2)] + \bar{x} \cdot [U \cdot (3 \cdot A_3 \cdot X^2 - 2 \cdot A_2 \cdot X + A_1)] - [U \cdot (A_3 \cdot X^3 - A_2 \cdot X^2 + A_1 \cdot X - A_0)]$

This is derived by:

$$\begin{aligned}
 [y] &= [y]_0 + [y]_1 \\
 &= \bar{u} \cdot ((\bar{a}_3 \cdot \bar{x}^3 + \bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) \\
 &\quad - (3 \cdot \bar{a}_3 \cdot \bar{x}^2 + 2 \cdot \bar{a}_2 \cdot \bar{x} + \bar{a}_1) \cdot [X] + (3 \cdot \bar{a}_3 \cdot \bar{x} + \bar{a}_2) \cdot [X^2] \\
 &\quad - \bar{a}_3 \cdot [X^3] - \bar{x}^3 \cdot [A_3] + \bar{x}^2 \cdot [3 \cdot A_3 \cdot X - A_2] \\
 &\quad - \bar{x} \cdot [3 \cdot A_3 \cdot X^2 - 2 \cdot A_2 \cdot X + A_1] \\
 &\quad + [A_3 \cdot X^3 - A_2 \cdot X^2 + A_1 \cdot X - A_0]) \\
 &\quad - (\bar{a}_3 \cdot \bar{x}^3 + \bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) \cdot [u]
 \end{aligned}$$



$$\begin{aligned}
 & + (3 \cdot \bar{a}_3 \cdot \bar{x}^2 + 2 \cdot \bar{a}_2 \cdot \bar{x} + \bar{a}_1) \cdot [u \cdot X] + \bar{x}^3 \cdot [u \cdot A_3] \\
 & - \bar{x}^2 \cdot [U \cdot (3 \cdot A_3 \cdot X - A_2)] \\
 & + \bar{x} \cdot [u \cdot (3 \cdot A_3 \cdot X^2 - 2 \cdot A_2 \cdot X + A_1)] \\
 & - [u \cdot (A_3 \cdot X^3 - A_2 \cdot X^2 + A_1 \cdot X - A_0)] \\
 = & \bar{u} \cdot (\bar{a}_3 \cdot \bar{x}^3 - 3 \cdot \bar{a}_3 \cdot \bar{x}^2 \cdot [X] + 3 \cdot \bar{a}_3 \cdot \bar{x} \cdot [X^2] - \bar{a}_3 \cdot [X^3] - \bar{x}^3 \cdot [A_3] \\
 & + 3 \cdot \bar{x}^2 \cdot [A_3 \cdot X] - 3 \cdot \bar{x} \cdot [A_3 \cdot X^2] + [A_3 \cdot X^3]) - \bar{a}_3 \cdot \bar{x}^3 \cdot [u] \\
 & + 3 \cdot \bar{a}_3 \cdot \bar{x}^2 \cdot [u \cdot X] - 3 \cdot \bar{a}_3 \cdot \bar{x} \cdot [U \cdot X^2] + \bar{a}_3 \cdot [U \cdot X^3] \\
 & + \bar{x}^3 \cdot [u \cdot A_3] - 3 \cdot \bar{x}^2 \cdot [u \cdot A_3 \cdot X] + 3 \cdot \bar{x} \cdot [U \cdot A_3 \cdot X^2] \\
 & - [U \cdot A_3 \cdot X^3] + \bar{u} \cdot ((\bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) \\
 & - (2 \cdot \bar{x} \cdot \bar{a}_2 + \bar{a}_1) \cdot [X] + \bar{a}_2 \cdot [X^2] \\
 & - \bar{x}^2 \cdot [A_2] + \bar{x} \cdot [2 \cdot A_2 \cdot X - A_1] - [A_2 \cdot X^2 - A_1 \cdot X + A_0]) \\
 & - (\bar{a}_2 + \bar{a}_1 \cdot \bar{x} + \bar{a}_0) \cdot \bar{x}^2 [u] + (2 \cdot \bar{x} \cdot \bar{a}_2 + \bar{a}_1) \cdot [u \cdot X] \\
 & - \bar{a}_2 \cdot [U \cdot X^2] + \bar{x}^2 \cdot [u \cdot A_2] - \bar{x} \cdot [u \cdot (2 \cdot A_2 \cdot X - A_1)] \\
 & + [U \cdot (A_2 \cdot X^2 - A_1 \cdot X + A_0)] \\
 = & \bar{u} \cdot (\bar{a}_3 \cdot \bar{x}^3 - 3 \cdot \bar{a}_3 \cdot \bar{x}^2 \cdot [X] + 3 \cdot \bar{a}_3 \cdot \bar{x} \cdot [X^2] - \bar{a}_3 \cdot [X^3] - \bar{x}^3 \cdot [A_3] \\
 & + 3 \cdot \bar{x}^2 \cdot [A_3 \cdot X] - 3 \cdot \bar{x} \cdot [A_3 \cdot X^2] + [A_3 \cdot X^3]) \\
 & - \bar{a}_3 \cdot \bar{x}^3 \cdot [u] + 3 \cdot \bar{a}_3 \cdot \bar{x}^2 \cdot [u \cdot X] - 3 \cdot \bar{a}_3 \cdot \bar{x} \cdot [U \cdot X^2] \\
 & + \bar{a}_3 \cdot [U \cdot X^3] + \bar{x}^3 \cdot [u \cdot A_3] - 3 \cdot \bar{x}^2 \cdot [u \cdot A_3 \cdot X] \\
 & + 3 \cdot \bar{x} \cdot [U \cdot A_3 \cdot X^2] - [U \cdot A_3 \cdot X^3] \\
 & + u \cdot (a_2 \cdot x^2 + a_1 \cdot x \cdot 2^p + a_0 \cdot 2^{2p}) \cdot 2^p \\
 = & (\bar{u} - [u]) \cdot (\bar{a}_3 \cdot \bar{x}^3 - 3 \cdot \bar{a}_3 \cdot \bar{x}^2 \cdot [X] + 3 \cdot \bar{a}_3 \cdot \bar{x} \cdot [X^2] - \bar{a}_3 \cdot [X^3] \\
 & - \bar{x}^3 \cdot [A_3] + 3 \cdot \bar{x}^2 \cdot [A_3 \cdot X] - 3 \cdot \bar{x} \cdot [A_3 \cdot X^2] + [A_3 \cdot X^3]) \\
 & + u \cdot (a_2 \cdot x^2 + a_1 \cdot x \cdot 2^p + a_0 \cdot 2^{2p}) \cdot 2^p \\
 = & (\bar{u} - [u]) \cdot (\bar{a}_3 - [A_3]) \cdot (\bar{x} - [X])^3 \\
 & + u \cdot (a_2 \cdot x^2 + a_1 \cdot x \cdot 2^p + a_0 \cdot 2^{2p}) \cdot 2^p \\
 = & u \cdot a_3 \cdot x^3 + u \cdot (a_2 \cdot x^2 + a_1 \cdot x \cdot 2^p + a_0 \cdot 2^{2p}) \cdot 2^p \\
 = & u \cdot (a_3 \cdot x^3 + a_2 \cdot x^2 \cdot 2^p + a_1 \cdot x \cdot 2^{2p} + a_0 \cdot 2^{3p})
 \end{aligned}$$

## B.2 Two-round ABY2.0-like evaluation

### B.2.1 For constant polynomials

Since sign-correcting a constant polynomial only requires a single multiplication as described in section B.1.1, a two-round ABY2.0-like evaluation is not required.

### B.2.2 For linear polynomials

**Precomputation:**  $P_2$  samples  $A_1$ ,  $U$ ,  $X$ , and  $Y$ , computes  $[A_1 \cdot X]$ ,  $[u \cdot Y]$ , and shares all six among  $P_0$  and  $P_1$ .

**Round 1:**  $P_b$  sends  $[\bar{a}_1]_b = [a_1]_b + [A_1]_b$  and  $[\bar{x}]_b = [x]_b + [X]_b$  to  $P_{1-b}$ , and vice versa.

**Round 2:**  $P_b$  sends  $[\bar{y}]_b = [y]_b + [Y]_b$  and  $[\bar{u}]_b = [u]_b + [U]_b$  to  $P_{1-b}$ , and vice versa. Here  $[y]_b = b \cdot (\bar{a}_1 \cdot \bar{x}) - \bar{a}_1 \cdot [X]_b - \bar{x} \cdot [A_1]_b + [A_1 \cdot X]_b + [a_0] \cdot 2^p$ .

**Output:**  $P_b$  outputs  $[u \cdot y]_b = b \cdot \bar{x} \cdot \bar{y} - \bar{x} \cdot [X]_b - \bar{y} \cdot [Y]_b + [X \cdot Y]_b$

The result of the first round of online communication is  $[y] = (\bar{a}_1 \cdot \bar{x}) - \bar{a}_1 \cdot [X] - \bar{x} \cdot [A_1] + [A_1 \cdot X] + [a_0] \cdot 2^p$

This is derived by:

$$\begin{aligned} [y] &= [y]_0 + [y]_1 \\ &= \bar{a}_1 \cdot \bar{x} - \bar{x} \cdot [A_1] - \bar{a}_1 \cdot [X] + [A_1 \cdot X] + [a_0] \cdot 2^p \\ &= (\bar{a}_1 - [A_1]) \cdot (\bar{x} - [X]) + [a_0] \cdot 2^p \\ &= a_1 \cdot x + a_0 \cdot 2^p \end{aligned}$$

The final Du-Atallah multiplication produces,  $u \cdot [y] = u \cdot (a_1 \cdot x + a_0 \cdot 2^p)$ , the sign corrected result.

### B.2.3 For quadratic polynomials

**Precomputation:**  $P_2$  samples  $A_1$ ,  $A_2$ ,  $U$ ,  $X$ , and  $Y$ , computes  $[X^2]$ ,  $[2 \cdot A_2 \cdot X - A_1]$ ,  $[A_2 \cdot X^2 - A_1 \cdot X]$ ,  $[u \cdot Y]$ , and shares all nine among  $P_0$  and  $P_1$ .

**Round 1:**  $P_b$  sends  $[\bar{a}_1]_b = [a_1]_b + [A_1]_b$ ,  $[\bar{a}_2]_b = [a_2]_b + [A_2]_b$ , and  $[\bar{x}]_b = [x]_b + [X]_b$  to  $P_{1-b}$ , and vice versa.

**Round 2:**  $P_b$  sends  $[\bar{y}]_b = [y]_b + [Y]_b$  and  $[\bar{u}]_b = [u]_b + [u]_b$  to  $P_{1-b}$ , and vice versa.  
 Here  $[y]_b = b \cdot (\bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x}) - \bar{x}^2 \cdot [A_2]_b + \bar{x} \cdot [2 \cdot A_2 \cdot X - A_1]_b - (\bar{a}_1 + 2 \cdot \bar{a}_2 \cdot \bar{x}) \cdot [X]_b + \bar{a}_2 \cdot [X^2]_b - [A_2 \cdot X^2 - A_1 \cdot X]_b + [a_0]_b \cdot 2^{2p}$ .

**Output:**  $P_b$  outputs  $[u \cdot y]_b = b \cdot \bar{x} \cdot \bar{y} - \bar{x} \cdot [X]_b - \bar{y} \cdot [Y]_b + [X \cdot Y]_b$

The result of the first round of online communication is  $[y] = (\bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x}) - \bar{x}^2 \cdot [A_2] + \bar{x} \cdot [2 \cdot A_2 \cdot X - A_1] - (\bar{a}_1 + 2 \cdot \bar{a}_2 \cdot \bar{x}) \cdot [X] + \bar{a}_2 \cdot [X^2] - [A_2 \cdot X^2 - A_1 \cdot X] + [a_0] \cdot 2^{2p}$

This value is derived by:

$$\begin{aligned}
 [y] &= [y]_0 + [y]_1 \\
 &= (\bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x}) - (2 \cdot \bar{a}_2 \cdot \bar{x} + \bar{a}_1) \cdot [X] + \bar{a}_2 \cdot [X^2] - \bar{x}^2 \cdot [A_2] \\
 &\quad + \bar{x} \cdot [2 \cdot A_2 \cdot X - A_1] - [A_2 \cdot X^2 - A_1 \cdot X] + [a_0] \cdot 2^{2p} \\
 &= (\bar{a}_2 \cdot \bar{x}^2 - 2 \cdot \bar{a}_2 \cdot \bar{x} \cdot [X] + \bar{a}_2 \cdot [X^2] - \bar{x}^2 \cdot [A_2] + 2 \cdot \bar{x} \cdot [A_2 \cdot X] \\
 &\quad - [A_2 \cdot X^2]) + \bar{a}_1 \cdot \bar{x} - \bar{x} \cdot [A_1] - \bar{a}_1 \cdot [X] + [A_1 \cdot X] \\
 &\quad + [a_0] \cdot 2^{2p} \\
 &= (\bar{a}_2 \cdot \bar{x}^2 - 2 \cdot \bar{a}_2 \cdot \bar{x} \cdot [X] + \bar{a}_2 \cdot [X^2] - \bar{x}^2 \cdot [A_2] + 2 \cdot \bar{x} \cdot [A_2 \cdot X] \\
 &\quad - [A_2 \cdot X^2]) + (a_1 \cdot x + a_0 \cdot 2^p) \cdot 2^p \\
 &= (\bar{a}_2 - [A_2]) \cdot (\bar{x} - [X])^2 + (a_1 \cdot x + a_0 \cdot 2^p) \cdot 2^p \\
 &= a_2 \cdot x^2 + a_1 \cdot x \cdot 2^p + a_0 \cdot 2^{2p}
 \end{aligned}$$

The final Du-Atallah multiplication produces the sign-corrected result  $u \cdot y = u \cdot (a_2 \cdot x^2 + a_1 \cdot x \cdot 2^{2p} + a_0 \cdot 2^p)$

#### B.2.4 For cubic polynomials

**Precomputation:**  $P_2$  samples  $A_1, A_2, A_3, U, X$ , and  $Y$ , computes  $[X^2]$ ,  $[X^3]$ ,  $[3 \cdot A_3 \cdot X - A_2]$ ,  $[3 \cdot A_3 \cdot X^2 - 2 \cdot A_2 \cdot X + A_1]$ ,  $[A_3 \cdot X^3 - A_2 \cdot X^2 + A_1 \cdot X]$ ,  $[u \cdot Y]$ , and shares all thirteen among  $P_0$  and  $P_1$ .

**Round 1:**  $P_b$  sends  $[\bar{a}_1]_b = [a_1]_b + [A_1]_b$ ,  $[\bar{a}_2]_b = [a_2]_b + [A_2]_b$ ,  $[\bar{a}_3]_b = [a_3]_b + [A_3]_b$ , and  $[\bar{x}]_b = [x]_b + [X]_b$  to  $P_{1-b}$ , and vice versa.

**Round 2:**  $P_b$  sends  $[\bar{y}]_b = [y]_b + [Y]_b$  and  $[\bar{u}]_b = [u]_b + [u]_b$  to  $P_{1-b}$ , and vice versa. Here  $[y]_b = b \cdot (\bar{a}_3 \cdot \bar{x}^3 + \bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x}) - (3 \cdot \bar{a}_3 \cdot \bar{x}^2 + 2 \cdot \bar{a}_2 \cdot \bar{x} + \bar{a}_1) \cdot [X]_b + (3 \cdot \bar{a}_3 \cdot \bar{x} + \bar{a}_2) \cdot [X^2]_b - \bar{a}_3 \cdot [X^3]_b - \bar{x}^3 \cdot [A_3]_b + \bar{x}^2 \cdot [3 \cdot A_3 \cdot X - A_2]_b - \bar{x} \cdot [3 \cdot A_3 \cdot X^2 - 2 \cdot A_2 \cdot X + A_1]_b + [A_3 \cdot X^3 - A_2 \cdot X^2 + A_1 \cdot X]_b + [a_0]_b \cdot 2^{3p}$ .

**Output:**  $P_b$  outputs  $[u \cdot y]_b = b \cdot \bar{x} \cdot \bar{y} - \bar{x} \cdot [X]_b - \bar{y} \cdot [Y]_b + [X \cdot Y]_b$

The result of the first round of online communication is  $[y] = (\bar{a}_3 \cdot \bar{x}^3 + \bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x}) - (3 \cdot \bar{a}_3 \cdot \bar{x}^2 + 2 \cdot \bar{a}_2 \cdot \bar{x} + \bar{a}_1) \cdot [X] + (3 \cdot \bar{a}_3 \cdot \bar{x} + \bar{a}_2) \cdot [X^2] - \bar{a}_3 \cdot [X^3] + \bar{x}^2 \cdot [3 \cdot A_3 \cdot X - A_2] - \bar{x} \cdot [3 \cdot A_3 \cdot X^2 - 2 \cdot A_2 \cdot X + A_1] + [A_3 \cdot X^3 - A_2 \cdot X^2 + A_1 \cdot X] + [a_0] \cdot 2^{3p}$ .

This value is derived by:

$$\begin{aligned}
 [y] &= [y]_0 + [y]_1 \\
 &= (\bar{a}_3 \cdot \bar{x}^3 + \bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x}) - (3 \cdot \bar{a}_3 \cdot \bar{x}^2 + 2 \cdot \bar{a}_2 \cdot \bar{x} + \bar{a}_1) \cdot [X] \\
 &\quad + (3 \cdot \bar{a}_3 \cdot \bar{x} + \bar{a}_2) \cdot [X^2] - \bar{a}_3 \cdot [X^3] - \bar{x}^3 \cdot [A_3] \\
 &\quad + \bar{x}^2 \cdot [3 \cdot A_3 \cdot X - A_2] - \bar{x} \cdot [3 \cdot A_3 \cdot X^2 - 2 \cdot A_2 \cdot X + A_1] \\
 &\quad + [A_3 \cdot X^3 - A_2 \cdot X^2 + A_1 \cdot X] + [a_0] \cdot 2^{3p} \\
 &= (\bar{a}_3 \cdot \bar{x}^3 - 3 \cdot \bar{a}_3 \cdot \bar{x}^2 \cdot [X] + 3 \cdot \bar{a}_3 \cdot \bar{x} \cdot [X^2] - \bar{a}_3 \cdot [X^3] \\
 &\quad - \bar{x}^3 \cdot [A_3] + 3 \cdot \bar{x}^2 \cdot [A_3 \cdot X] - 3 \cdot \bar{x} \cdot [A_3 \cdot X^2] + [A_3 \cdot X^3]) \\
 &\quad + (\bar{a}_2 \cdot \bar{x}^2 + \bar{a}_1 \cdot \bar{x}) - (2 \cdot \bar{a}_2 \cdot \bar{x} + \bar{a}_1) \cdot [X] + \bar{a}_2 \cdot [X^2] \\
 &\quad - \bar{x}^2 \cdot [A_2] + \bar{x} \cdot [2 \cdot A_2 \cdot X - A_1] - [A_2 \cdot X^2 - A_1 \cdot X] \\
 &\quad + [a_0] \cdot 2^{3p} \\
 &= (\bar{a}_3 \cdot \bar{x}^3 - 3 \cdot \bar{a}_3 \cdot \bar{x}^2 \cdot [X] + 3 \cdot \bar{a}_3 \cdot \bar{x} \cdot [X^2] - \bar{a}_3 \cdot [X^3] \\
 &\quad - \bar{x}^3 \cdot [A_3] + 3 \cdot \bar{x}^2 \cdot [A_3 \cdot X] - 3 \cdot \bar{x} \cdot [A_3 \cdot X^2] + [A_3 \cdot X^3]) \\
 &\quad + (a_2 \cdot x^2 + a_1 \cdot x \cdot 2^p + a_0 \cdot 2^{2p}) \cdot 2^p \\
 &= (\bar{a}_3 - [A_3]) \cdot (\bar{x} - [X])^3 + (a_2 \cdot x^2 + a_1 \cdot x \cdot 2^p + a_0 \cdot 2^{2p}) \cdot 2^p \\
 &= a_3 \cdot x^3 + a_2 \cdot x^2 \cdot 2^p + a_1 \cdot x \cdot 2^{2p} + a_0 \cdot 2^{3p}
 \end{aligned}$$

The final Du-Atallah multiplication produces the sign-corrected result  $u \cdot y = u \cdot (a_3 \cdot x^3 + a_2 \cdot x^2 \cdot 2^p + a_1 \cdot x \cdot 2^{2p} + a_0 \cdot 2^{3p})$

## B.3 Horner's Method evaluation

### B.3.1 For linear polynomials

Identical to the two-round ABY2.0-like linear evaluation.

**Precomputation:**  $P_2$  samples  $A_1, U, X$ , and  $Y$ , computes  $[A_1 \cdot X]$  and  $[u \cdot Y]$ , and shares all six among  $P_0$  and  $P_1$ .

**Round 1:** Du-Atallah multiply  $a_1 \cdot x$ . To do this,  $P_b$  sends  $\bar{a}_1 = [a_1] + [A_1]$  and  $\bar{x} = [x] + [X]$  to  $P_{1-b}$ , and vice versa.

**Round 2:** Du-Atallah multiply  $u \cdot y_1$  where  $y_1 = a_1 \cdot x + a_0 \cdot 2^p$ . To do this,  $P_b$  sends  $[\bar{u}]_b = [u]_b + [u]_b$  and  $[\bar{y}]_b = [y]_b + [Y]_b$  to  $P_{1-b}$ , and vice versa.

**Output:**  $P_b$  outputs  $u \cdot y_1 = u \cdot (a_1 \cdot x + a_0 \cdot 2^p)$

Total:

- 6 precomputed values
- 2 rounds
- 4 values sent online by each of  $P_0$  and  $P_1$

### B.3.2 For quadratic polynomials

**Precomputation:**  $P_2$  samples  $A_2, U, X, Y_1$ , and  $Y_2$ , computes  $[A_2 \cdot X]$ ,  $[Y_1 \cdot X]$ , and  $[u \cdot Y_2]$ , and shares all eight among  $P_0$  and  $P_1$ .

**Round 1:** Du-Atallah multiply  $a_2 \cdot x$ . To do this,  $P_b$  sends  $\bar{a}_2 = [a_2] + [A_2]$  and  $\bar{x} = [x] + [X]$  to  $P_{1-b}$ , and vice versa.

**Round 2:** Du-Atallah multiply  $[y_1] \cdot [x]$  where  $y_1 = a_2 \cdot x + a_1 \cdot 2^p$ . To do this,  $P_b$  sends  $[\bar{y}_1] = [y_1] + [Y_1]$  to  $P_{1-b}$ , and vice versa.

**Round 3:** Du-Atallah multiply  $u \cdot y_2$  where  $y_2 = y_1 \cdot x + a_0 \cdot 2^{2p} = (a_2 \cdot x + a_1 \cdot 2^p) \cdot x + a_0 \cdot 2^{2p}$ . To do this,  $P_b$  sends  $[\bar{u}]_b = [u]_b + [u]_b$  and  $[\bar{y}]_b = [y]_b + [Y]_b$  to  $P_{1-b}$ , and vice versa.

**Output:**  $P_b$  outputs  $u \cdot y = u \cdot (a_2 \cdot x^2 + a_1 \cdot x \cdot 2^p + a_0 \cdot 2^{2p})$

Result:  $u \cdot ((a_2 \cdot x + a_1) \cdot x + a_0)$

Total:

- 8 precomputed values
- 3 rounds
- 5 values sent online by each of  $P_0$  and  $P_1$

### B.3.3 For cubic polynomials

**Precomputation:**  $P_2$  samples  $A_3, U, X, Y_1, Y_2$ , and  $Y_3$ , computes  $[A_3 \cdot X], [Y_1 \cdot X], [Y_2 \cdot X]$ , and  $[u \cdot Y_3]$ , and shares all ten among  $P_0$  and  $P_1$ .

**Round 1:** Du-Atallah multiply  $a_3 \cdot x$ . To do this,  $P_b$  sends  $\bar{a}_3 = [a_3] + [A_3]$  and  $\bar{x} = [x] + [X]$  to  $P_{1-b}$ , and vice versa.

**Round 2:** Du-Atallah multiply  $[y_1] \cdot [x]$  where  $y_1 = a_3 \cdot x + a_2 \cdot 2^p$ . To do this,  $P_b$  sends  $[\bar{y}_1] = [y_1] + [Y_1]$  to  $P_{1-b}$ , and vice versa.

**Round 3:** Du-Atallah multiply  $[y_2] \cdot [x]$  where  $y_2 = (a_3 \cdot x + a_2 \cdot 2^p) \cdot x + a_1 \cdot 2^{2p}$ . To do this,  $P_b$  sends  $[\bar{y}_2] = [y_2] + [Y_2]$  to  $P_{1-b}$ , and vice versa.

**Round 4:** Du-Atallah multiply  $u \cdot y_3$  where  $y_2 = y_2 \cdot x + a_0 \cdot 2^{3p} = ((a_3 \cdot x + a_2 \cdot 2^p) \cdot x + a_1 \cdot 2^{2p}) \cdot x + a_0 \cdot 2^{3p}$ . To do this,  $P_b$  sends  $[\bar{u}]_b = [u]_b + [u]_b$  and  $[\bar{y}]_b = [y]_b + [Y]_b$  to  $P_{1-b}$ , and vice versa.

**Output:**  $P_b$  outputs  $u \cdot y = u \cdot (a_3 \cdot x^3 + a_2 \cdot x^2 \cdot 2^p + a_1 \cdot x \cdot 2^{2p} + a_0 \cdot 2^{3p})$ .

Result:  $u \cdot (y_2 \cdot x + a_0 = ((a_3 \cdot x + a_2) \cdot x + a_1) \cdot x + a_0)$

Total:

- 10 precomputed values
- 4 rounds
- 6 values sent online by each of  $P_0$  and  $P_1$

## Appendix C

### Proof of Theorem 10 (from Section 7.3)

**Theorem 10 (Restatement).** *The 3-verifier SNIP auditing protocol with hash function  $\text{Hash}: \{0, 1\}^* \rightarrow \{0, 1\}^\mu$  is perfectly simulatable and has perfect completeness and soundness overwhelming in  $\mu$ .*

*Proof (sketch).* Perfect simulatability and completeness are immediate consequences of the perfect simulatability and completeness of  $(2 + 1)$ -party auditing. To see that soundness is indeed overwhelming in  $\mu$ , it suffices to note that since the three verifiers collectively scrutinize all three parties' views from a simulated  $(2 + 1)$ -party audit, inconsistencies in the simulation will never escape notice by at least one verifier—except, perhaps, in the event of a hash collision when constructing the Merkle tree. In other words, in the absence of hash collisions, soundness is perfect and, consequently, the overall soundness error is negligible in  $\mu$ .  $\square$

## Appendix D

# Proof of Theorem 11 (from Section 7.3)

**Theorem 11 (Restatement).** *The 2-verifier SNIP auditing protocol with hash function  $\text{Hash}: \{0, 1\}^* \rightarrow \{0, 1\}^\mu$  is perfectly simulatable and has perfect completeness and soundness overwhelming in  $\mu$ .*

*Proof (sketch).* Perfect simulatability and completeness are immediate consequences of the perfect simulatability and completeness of the 2-verifier SNIPs (which, in turn, are an immediate consequence of the perfect simulatability and completeness of  $(2 + 1)$ -party auditing). To prove that soundness is overwhelming in  $\mu$ , we note that even a single inconsistency-free simulation among the  $\lambda$  parallel simulations is sufficient to establish well-formedness of the DPF with probability overwhelming in  $\mu$ ; thus, we can assume without loss of generality that *all* simulations are inconsistent to get an upper bound on soundness error. For a given inconsistent simulation, there are two possibilities:

1. inconsistencies are confined to the view of server 2; or
2. inconsistencies exist in the view of server 0 or server 1 (or both).

In either case, at least one verifier will observe any inconsistencies in the  $i$ th parallel simulation with probability at least  $\frac{1}{2}$  (i.e., conditioned on the  $i$ th challenge bit,  $c_i$ ),



yielding an overall soundness loss of at most  $1/2^\lambda$  relative to the 3-verifier SNIP, which has soundness error negligible in  $\mu$ .  $\square$

## Appendix E

# Stepping stones to Sabre-M

This appendix describes the “stepping stones” that led to the development Sabre-M.

### Sabre-M0: Express with improved auditing

Consider a Sabre-M0 instance with security parameter  $\lambda \in \mathbb{N}$  (say,  $\lambda = 128$ ). The design of Sabre-M0 tightly parallels that of Express, save for the new audit protocol. In a registration phase, the recipient contacts the servers to create a mailbox and receives a uniform random,  $\lambda$ -bit address in exchange. Suppose there are  $n$  registered mailboxes with addresses  $addr_1, \dots, addr_n$ . To deposit a message  $M \in \mathbb{F}_{2^L}$  into the mailbox addressed by  $addr_i$ , the sender samples  $(\mathbf{dpf}[0], \mathbf{dpf}[1]) \leftarrow \text{Gen}(1^\lambda, \mathbb{F}_{2^\lambda}, \mathbb{F}_{2^L}; addr_i, M)$  and then it sends  $\mathbf{dpf}[b]$  to server  $b$  for  $b = 0, 1$ .

Upon receiving  $\mathbf{dpf}[b]$ , server  $b$  constructs the vector  $M_b \in (\mathbb{F}_{2^L})^{1 \times n}$  in which the  $j$ th component equals  $\text{Eval}(\mathbf{dpf}[b], addr_j)$ ; server  $b$  adds  $M_b$  to its mailbox database to effectuate the write. We stress that (as in Express—owing to the sparsity of the set of mailbox addresses within  $\{0, 1\}^\lambda$ ) Boyle et al.’s *full-domain evaluation* procedure does not apply when computing  $M_b$ . Instead, each column requires  $O(\lambda)$  length-doubling PRG evaluations (for a total of  $O(\lambda n)$  evaluations), plus one evaluation of the leaf-stretching PRG.

As per [Definition 4](#),  $M_0 + M_1 \in (\mathbb{F}_{2^L})^{1 \times n}$  has message  $M$  in its  $i$ th column and zero elsewhere—*provided the sender generated  $(\mathbf{dpf}[0], \mathbf{dpf}[1])$  honestly*, which the servers confirm via any of the three audit protocol instantiations. Sabre-M0 very quickly rejects malformed DPFs outright, whereas (like in Express) otherwise well-

formed DPFs whose distinguished points do not correspond to registered mailbox addresses will pass auditing, ultimately resulting in a (comparatively expensive) no-operation “update” to the mailbox database. In particular, such well-formed, yet bogus writes incur the same *decidedly non-trivial* cost as “true” writes.

### Sabre-M1: Sabre-M0 with shorter DPFs

Sabre-M1 provides a modest improvement over Sabre-M0 by dynamically “truncating” mailbox addresses to a prefix just long enough to avoid collisions with high probability, thereby reducing both (i) the size of (and cost of auditing) the keys  $\mathbf{dpf}[0]$  and  $\mathbf{dpf}[1]$  and (ii) the cost of computing  $M_0$  and  $M_1$  from them.

Upon registering a mailbox, the recipient still receives a uniform random,  $\lambda$ -bit mailbox address that senders must know (in its entirety) to deposit messages into the mailbox. The difference is that now only a prefix of this address (whose length grows with the number of registered mailboxes) is used when sampling and evaluating DPFs. The intuition here is that—for purposes of correctness—it suffices merely to guarantee that no two mailboxes map to the same prefix; thus, one can compute the required prefix length for a given number of registered mailboxes using a standard birthday bound calculation. (In our experiments in [Section 7.6.4](#), we chose the length to ensure that the probability of one or more prefix collisions is strictly less than 0.001.)

This optimization improves efficiency relative to Sabre-M0. However, it comes at the cost of increasing the probability of an attacker “guessing” a distinguished point corresponding to a valid mailbox—that is, random guesses potentially clobber a registered mailbox with a probability that is now *polynomial in  $n$* . To mitigate, Sabre-M1 requires the sender to transmit *an additive share*  $[addr]$  of the full mailbox address alongside  $\mathbf{dpf}[b]$ , after which the servers use a simple PIR-based protocol to verify that the address shares reconstruct to the full address of whatever mailbox the DPF keys reference.

Specifically, servers 0 and 1 hold in common an associative array (key-value store) mapping each distinguished input to its corresponding mailbox addresses. Let  $D$  denote the set of distinguished inputs reflected in the associative array and, for each  $i \in D$ , let  $\text{flag}_b^{(i)}$  denote the *advice bit* for the  $i$ th leaf of  $\mathbf{dpf}[b]$  (see [Section 7.2.1](#)).

By construction, if  $(\mathbf{dpf}[0], \mathbf{dpf}[1])$  is a valid DPF key pair, then  $\text{flag}_0^{(i)} = \text{flag}_1^{(i)}$  if and only if the distinguished point is  $i$ ; thus, server  $b$  outputs

$$[\text{addr}']_b := \sum_{i \in D, t_i^{(b)}=1} \text{addr}_i$$

so that  $[\text{addr}']_0 + [\text{addr}']_1 = \text{addr}_i$  over  $\mathbb{F}_{2^\lambda}$ . The servers verify that this *computed* address  $[\text{addr}']$  matches the client's *claimed* address  $[\text{addr}]$  by checking that  $[\text{addr}]_0 + [\text{addr}']_0 = [\text{addr}]_1 + [\text{addr}']_1$ .

The *full-domain evaluation* procedure still does not apply when computing  $M_b$  in Sabre-M1; however, now the cost of computing each column reduces from  $O(\lambda)$  to just  $O(\text{poly}(\lg n))$  length-doubling PRG evaluations (reducing the total cost of computing  $M_b$  from  $O(n\lambda)$  to just  $O(n\text{poly}(\lg n))$  evaluations), plus  $n$  evaluations of the leaf-stretching PRG. Furthermore, the servers can now detect and reject DPFs from senders who do not know a registered mailbox address *without incurring a costly no-operation to “update” the database*.

### Sabre-M2: Sabre-M1 with decoupled address checking

Sabre-M2 improves on Sabre-M1 by eliminating the association between DPF leaves and mailbox addresses: Instead of sampling DPFs whose distinguished points correspond to mailbox addresses, senders sample DPFs whose distinguished points correspond to the *chronological order in which those mailboxes were registered*. In other words, to deposit a message  $M \in \mathbb{F}_{2^L}$  into the mailbox with address  $\text{addr}_i$ , a sender samples keys for a DPF with point  $(i, M) \in [0 \dots n-1] \times \mathbb{F}_{2^L}$  rather than  $(\text{addr}_i, M) \in \mathbb{F}_{2^\lambda} \times \mathbb{F}_{2^L}$ . With this optimization, (i) the domain of DPFs has size equal to the number of registered mailboxes, and (ii) the *full-domain evaluation* procedure applies when computing the  $M_b$ . This reduces the total cost of computing  $M_b$  to just  $n-1$  length-doubling PRG evaluations plus  $n$  evaluations of the leaf-stretching PRG. To ensure that senders know the correct mailbox address, Sabre-M2 inherits the PIR-based address checking of Sabre-M1.

### Sabre-M: Sabre-M2 with $O(1)$ -time address checking

Sabre-M improves on Sabre-M2 by replacing the linear-complexity PIR-based address check with a constant-complexity *PRF-based* one described in [Section 7.5.2](#).

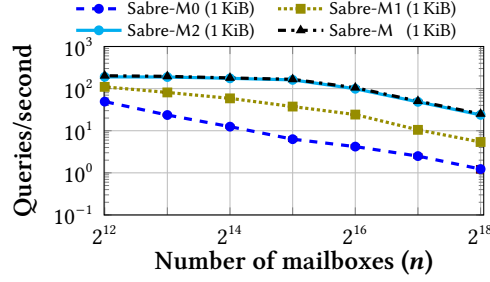


Figure E.1: Throughput of Sabre-M variants (1 KiB messages)

With this modification in place, the entire auditing procedure—that is, both checking the well-formedness of the DPF *and* checking the validity of the mailbox address—has complexity logarithmic in  $n$ , allowing the servers to rapidly reject bogus write requests from a would-be resource-exhaustion DoS attacker. The benefits of rapid auditing are shown in our experimental evaluation in [Section 7.6](#).

[Figure E.1](#) plots the throughput of each Sabre-M variant. Notice that the plots for Sabre-M2 and Sabre-M overlap almost perfectly; we stress that this neck-to-neck performance is an artifact of all writes being valid.

## Appendix F

# Sabre Security Analysis

This appendix provides a formal definition and proof sketch for Sabre in the ideal-world/real-world simulation paradigm. In the ideal world, all Sabre clients interact through some benevolent trusted party  $\mathcal{T}$  (the “ideal functionality”), who faithfully executes the requested actions while remaining impervious to cryptographic attacks. In the real world, we replace  $\mathcal{T}$  with one of the 2- or 3-server Sabre protocols described in the main text. We then consider a semi-honest attacker  $\mathcal{A}$  who controls an arbitrary number of readers and writers in addition to (at most) one of the Sabre servers.

Informally, we wish to show that  $\mathcal{A}$  cannot exploit its privileged position as a Sabre server to compromise sender anonymity. We do this by exhibiting an efficient simulator that interacts with the ideal functionality and then attempts to sample “simulated” views from a distribution close to the one describing  $\mathcal{A}$ ’s view in the real world. We then ask whether  $\mathcal{A}$  can adaptively conjure up sequences of events allowing it to distinguish between real and simulated views; if not, we conclude that the real Sabre protocols leak essentially nothing beyond what is leaked by their ideal-world counterparts.

### Sabre in the ideal world

The ideal world trusted party  $\mathcal{T}$  exposes six public interfaces. In addition to servicing requests to these interfaces,  $\mathcal{T}$  curates persistent variables that capture both the state of the system and any side information that Sabre servers can infer in the real world

(without  $\mathcal{T}$  revealing this side information to the simulator, faithful simulations would not be possible):

- $m, w \in \mathbb{N}$  respectively track the numbers of readers and writers in the system (both values are initially 0);
- $addr_j$  denotes the mailbox address/bucket index at which reader  $j$  will check for messages
- $known$  denotes the sets of readers whose mailbox addresses/bucket indexes have been *disclosed to*  $\mathcal{A}$  (it is initially  $\emptyset$ );
- $evil$  denotes the set of “corrupted” readers whose mailboxes are *readable by*  $\mathcal{A}$  (it is initially  $\emptyset$ );
- $log$  is an append-only log of metadata associated with all requests received by  $\mathcal{T}$  to date (it is initially an empty list); and
- $D$  is the actual database of mailboxes/buckets (it is initially an empty list).

The RegisterWriter interface onboards a new writer to the system. This interface has no immediate counterpart in real world; rather, it captures the fact that real-world Sabre servers may be able to distinguish certain writers from one another (e.g., by their IP addresses).

---

**Algorithm 3** RegisterWriter()

---

1:	$w \leftarrow w + 1$	<i>// increment writer count</i>
2:	<b>return</b> $w$	<i>// w is the writer’s “identity”</i>

---

The RegisterReader interface onboards new readers to the system. In the bulletin-board model, this interface has no immediate counterpart in the real world; rather, it captures the fact that readers and writers need to agree on a bucket through which to pass messages. In the mailbox model, this interface corresponds to mailbox registration.

**Algorithm 4** RegisterReader()

---

```

1:  $m \leftarrow m + 1$  // increment reader count
2: if Sabre-BB then
    $addr_m \xleftarrow{?} \{0, 1\}^n$  // random address
    $evil \leftarrow evil \cup \{m\}$  // buckets are world-readable
3: else if Sabre-M then
    $addr_m := F(\tilde{k}, m)$  // PRF-computed address
    $known \leftarrow known \cup \{m\}$  // server-computable addresses
4: end if
5:  $log \leftarrow log || "r:m"$ 
6: return  $m$  // m is the reader's identity

```

---

The GetAddress interface allows  $\mathcal{A}$  to request the mailbox address (or bucket index) of a given reader. Note that we differentiate between  $\mathcal{A}$  *controlling* a reader versus merely *knowing its mailbox address*: To write messages to a mailbox,  $\mathcal{A}$  need only know its address; to read from the mailbox,  $\mathcal{A}$  must control the reader.

**Algorithm 5** GetAddress( $j$ )

---

```

1: if ( $j > m$ ) then
   return  $\perp$  // no such reader!
2:  $known \leftarrow known \cup \{j\}$  // add j to known addresses
3: return  $addr_j$  // then disclose j's address

```

---

The CorruptReader interface allows  $\mathcal{A}$  to take control of an arbitrary reader (whose address  $\mathcal{A}$  already knows).

**Algorithm 6** CorruptReader( $j$ )

---

```

1: if ( $j \notin known$ ) then
   return  $\perp$  // must get address first!
2:  $evil \leftarrow evil \cup \{j\}$  // add j to corrupt list

```

---

The Write interface allows  $\mathcal{A}$  to write a message for a given reader, provided it knows that reader's address. This interface also allows  $\mathcal{A}$  to initiate write requests that fail address verification or auditing (by setting  $d < h$ ).



**Algorithm 7** Write( $i, j, addr, M, d [= h]$ )

---

```

1: if ( $i > w$ ) then
    return  $\perp$                                 // no such writer
2: if ( $j > m \vee addr \neq addr_j$ ) then
     $log \leftarrow log || "w: (i, \perp)"$         // address check failed
    return  $\perp$                                 // abort without writing
3: if ( $d \neq h$ ) then
     $log \leftarrow log || "w: (i, d)"$           // audit failed at level d
    return  $\perp$                                 // abort without writing
4:  $log \leftarrow log || "w: (i, \top)"$         // this is a valid write
5:  $D_j \leftarrow D_j + M$                     // effectuate the write

```

---

The EndEpoch interface ends the current epoch, writing all corrupted readers' mailbox/bucket contents to the log.

**Algorithm 8** EndEpoch()

---

```

1:  $msgs \leftarrow \{(j, D_j) \mid j \in evil\}$     // corrupted mailbox contents
2:  $D \leftarrow \langle 0, 0, \dots, 0 \rangle$           // re-initialize the database
3:  $log \leftarrow log || "e: msgs"$             // new epoch after this point

```

---

**Security theorem**

We now present and sketch a proof for Sabre-M's main security theorem (the theorems and proofs for Sabre-BB and Sabre-M0 through Sabre-M2 are similar). Intuitively, the theorem asserts that there exists a PPT simulator that, given only the append-only  $log$  and PRF key  $\tilde{k}$  as input, can convincingly simulate the view of an arbitrary semi-honest Sabre server.

**Theorem 12** (somewhat informal). *Sabre is a secure SAM scheme; that is, there exists a PPT simulator that, given the PRF key  $\tilde{k}$  and side information reflected in  $log$ , simulates the view of a (semi-honest, PPT) real-world attacker  $\mathcal{A}$  controlling at most one Sabre server and an arbitrary subset of writers and readers.*

*Proof (sketch).* There are 5 distinct types of entries that can appear on  $log$ ; we briefly describe how the simulator deals with each of these five message types in turn.

1. "**r:m**": This entry appears in the log when the  $m$ th reader registers her mailbox.

In Sabre-M, mailbox registration consists of a server evaluating  $F(\tilde{k}, m)$  and then sending the result to the registrant; hence, the simulator (who is privy to  $\tilde{k}$ ) need only invoke  $addr_m \leftarrow F(\tilde{k}, m)$  and record the resulting mailbox address in its simulation transcripts.

2. “w: (i,  $\perp$ )”: This entry appears in the log when reader  $i$  submits a write request that fails the mailbox address check.

In Sabre-M, the servers verify addresses by computing  $[addr'] \leftarrow F(\tilde{k}, [i])$  and then comparing it against  $[addr]$ ; hence, the simulator (who is privy to  $\tilde{k}$ ) first invokes the simulator for the DPFs to sample a valid-looking DPF key, and then it samples uniform random values for  $[i]_b$  and  $[addr]_b$ , and then it simulates the computation of  $[addr] \leftarrow F(\tilde{k}, [i])$ . Finally, it computes  $A_b = [addr]_b + [addr']_b$ , samples a uniform random  $A_{1-b} \neq A_b$ , and then records all of these values in its simulation transcript.

3. “w: (i,  $d$ )”: This entry appears in the log when reader  $i$  submits a write request that passes the mailbox check but fails auditing at level  $d$ .

The simulator invokes the simulator for the DPF to produce a valid-looking DPF key, and then it repeatedly invokes the simulator for Du-Atallah multiplication to construct an accepting proof up to level  $d - 1$ . In Step 2 of level  $d$ , it samples uniform random  $v_0^{(i_d)}$  and  $v_1^{(i_d)}$  with  $v_0^{(i_d)} \neq v_1^{(i_d)}$  (causing the servers to **reject**), and then it records these values in its simulation transcript.

4. “w: (i,  $\top$ )”: This entry appears in the log when reader  $i$  successfully writes a message to some reader.

The simulator proceeds exactly as in the case of “w: (i,  $d$ )”, except it does not **reject** along the way and instead **accepts** after level  $h$ . It then invokes the full-domain evaluation on its (simulated) DPF key and adds the resulting vector component-wise into its mailbox database.

5. “e: *msgs*”: This entry appears in the log when the epoch ends (at which point all readers receive any messages that were written into their mailboxes).

The simulator uses the messages in *msgs* together with its simulated mailbox database to solve for the matching shares from server  $b - 1$  and then it records these values in its simulation transcripts for each corrupted reader.

□

## Appendix G

### LowMC parameter selection

LowMC contains several tuning knobs allowing tradeoffs among the total number of multiplications (i.e., s-boxes per round), multiplicative depth (number of rounds), and wall-clock running time. Roughly speaking, optimizing for total number of multiplications minimizes overall communication cost (for MPC-based auditing) and proof size (for SNIP-based auditing), whereas optimizing for multiplicative depth minimizes round complexity (for MPC-based auditing) or proof verification time (for SNIP-based auditing). For our Sabre implementation, we chose to optimize for multiplicative depth by maximizing the number of s-boxes per round. [Figure G.1](#) presents empirical evidence to justify this choice; specifically, [Figures G.1a–G.1c](#) respectively plot the wall-clock time for 2-verifier SNIP auditing, the wall-clock running time for full-domain DPF evaluation, and the expected size of 2-verifier SNIPs for various settings of the “s-boxes per round” tuning knob and numbers of buckets/mailboxes. The graphs show that optimizing for multiplicative depth is essentially pessimal for auditing but optimal for full-domain evaluation, which is the bottleneck operation in all Sabre variants.

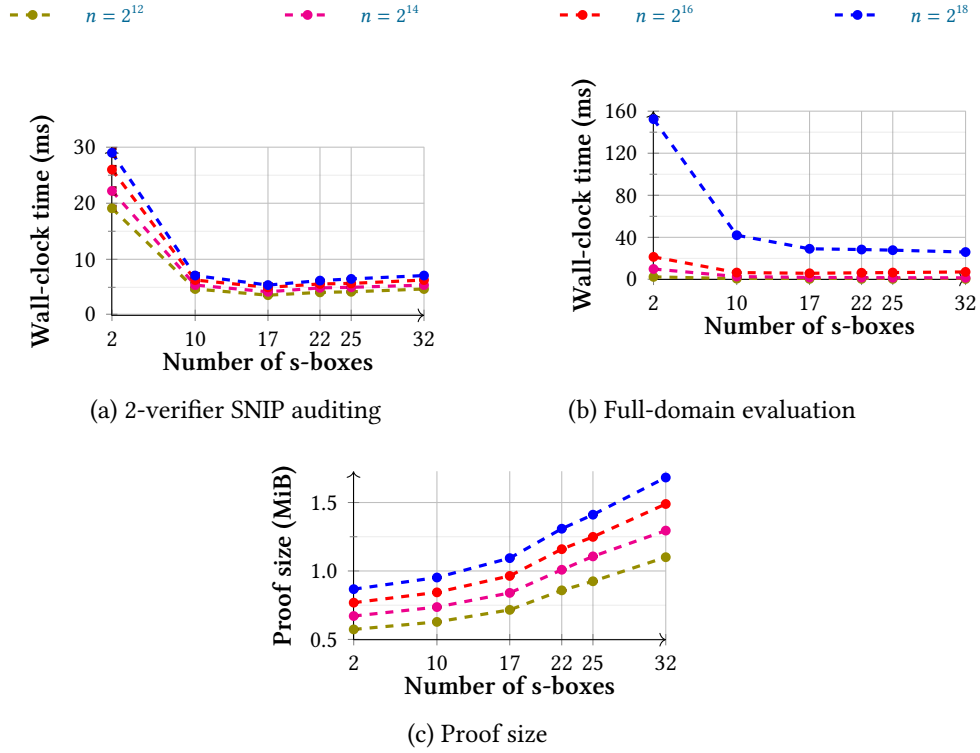


Figure G.1: 2-verifier SNIP auditing time, full-domain evaluation time, and proof size versus number of LowMC s-boxes.

## Appendix H

### Sabre Communication Cost

In this section, we will present the calculations that lead to the communication cost described in [Table 7.1](#) and [Table 7.2](#).

We first calculate the size of  $M_i^{b \rightarrow (1-b)}$ . The transcript from server 0 to server 1 include the following (the transcript of server 1 to server 0 is the same).

(i) shares of the children of the root = 16 bytes, (ii) oblivious encryption,  $F_{PRG} = (4 \cdot 3 \cdot s \cdot r/8)(h-1)$  bytes, (iii) conditional swap,  $F_{SWAP} = 16 \cdot h$  bytes, (iv) applying the correction word =  $16 \cdot (h-1)$  bytes, and (v) computing the next flag bit  $(1/8 + 1/8) \cdot (h-1)$  bytes. size of  $M_i^{b \rightarrow (1-b)} = 16 + (16\frac{1}{8} + \frac{3}{2}sr)(h-1)$  Next, we calculate the size  $M_i^{2 \rightarrow b}$  for  $b \in \{0, 1\}$ . (i) seed to generate the randomness = 16 bytes, (ii) oblivious encryption,  $F_{PRG} = (4 \cdot 3 \cdot s \cdot r/8)(h-1)$ , (iii) applying the correction word =  $16 \cdot (h-1)$  bytes, (iv) conditional swap,  $F_{SWAP} = 16 \cdot h$  bytes, and (v) computing the next flag bit  $(1/8) \cdot (h-1)$  bytes. Size of  $M_i^{b \rightarrow (1-b)} = 16 + (32\frac{1}{8} + \frac{3}{2}sr)(h-1)$ .