On Efficient Instantiations of Secure Multi-Party Computation in Practice

by

Alexander Bienstock

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

January, 2024

Dedication

To Sarah, with love.

Acknowledgments

First, I would like to thank Yevgeniy Dodis, who took me on as a student at NYU and brought to me several interesting problems that would end up being the basis of my first research papers. Yevgeniy always pushed me to go beyond my comfort zone, and for that, I am thankful. I would also like to thank Marshall Ball, who only came to NYU at the start of my third year, but introduced me to new and exciting areas of cryptography and kindly advised me for the rest of my studies. I always look forward to the fun and low stress, but also productive, meetings I have with Marshall. To the rest of my committee — Joe Bonneau, Sanjam Garg, and Antigoni Polychroniadou — thank you for joining me for the end of this journey.

Next, I would like to thank several of my other mentors throughout the years. Allison Bishop was the one who introduced me to the wonderful world of cryptography in my undergraduate studies. Only through Allison was I able to discover my love for cryptography. Paul Rösler gave me much needed advice on how to actuallly write a research paper when we were collaborating on my first paper. He has been a valued coauthor ever since. My summer working with Sanjam Garg at Berkeley in 2021 was a turning point in my PhD. I was able to truly find my footing as a researcher with Sanjam and for that, I am forever grateful. Sanjam spent hours working with me and was always there for me even when I had the most trivial of questions. The summer of 2022 was when I discovered my passion for MPC. This would not have been possible without the support of Daniel Escudero and Antigoni Polychroniadou at JPMorgan. I learned the basics of MPC from Antigoni and Daniel and could always count on them to answer my various questions.

Finally, I would like to thank Kevin Yeo for mentoring me at Google in the Fall of 2022. Kevin revealed to me many of the aspects of cryptography in practice which were unbenknownst to me beforehand. Kevin was also always open to me about his current thoughts on several research problems.

I would now like to thank all of the wonderful coauthors I have had throughout the years — Jaiden Fairoze, Garrison Grogan, Mohammad Hajiabadi, Lisa Kohl, Pierre Meyer, Pratyay Mukherjee, Sarvar Patel, Srinivasan Raghuraman, Joon Young (Mike) Seo, and Yi Tang. I would also like to express my gratitude for the support that I received from the members of the various research groups of which I have been a part (who I have not mentioned above). At NYU — Arasu Arun, Eli Goldin, Peter Hall, Daniel Jost, and Harish Karthikeyan. At Berkeley/NTT — James Bartusek, Alex Lombardi, Fermi Ma, and Willy Quach. At JPMorgan — Laasya Bangalore, Yue Guo, Eysa Lee, Yiping Ma, Lisa Masserova, Sahar Mazloom, Nikolas Melissaris, and Chenkai Weng. At Google — Ben Kreuter, Mariana Raykova, Phillip Schoppmann, Karn Seth, and Moti Yung.

I am also very grateful for the support that my family — Mom, Dad, Dennis, and Doug — and various friends have given me throughout the PhD process, even if they could not always understand what I was doing or why I was doing it.

Last but not least I would like to thank my girlfriend, Sarah, to whom this thesis is dedicated. As you well know, the PhD has been a rollercoaster of ups and downs for me, and you have been a solid anchor of support throughout. After my greatest achivements, you always gave me the most enthusiastic congratulations; during my biggest struggles, you always assured me that everything would be okay. Without you, none of this would have been possible.

Abstract

Secure Multi-Party Computation (MPC) is an area of cryptography that has been studied extensively since the 1980s. In full generality, MPC allows a set of mutually distrusting parties to *privately* compute a function of their inputs. That is, the parties interact in some protocol, and at the end obtain the output of the function, and nothing else. In the decades since the inception of MPC, great strides have been made towards making it more efficient. However, despite this progress, the use of MPC in practice still faces some shortcomings.

In this thesis, we take steps to mitigate two such shortcomings. The first deficiency we study is related to the communication networks in which such MPC protocols operate. MPC protocols are usually designed assuming that all parties have pairwise secure communication channels which are *stable*; i.e., nodes never crash, messages always arrive on time, etc. However, in the real-world, this is rarely the case—it is hard to sustain a stable connection between parties over long periods of time. One such model that has been introduced to address this deficiency is called *Fluid MPC* (Choudhuri *et al.*, CRYPTO 2021). In this model, parties are not mandated to stay online for long periods of time. Instead, parties come online for short periods of time and work together in *committees* to compute some function. The benefit is that individual committees are much more likely to be able to sustain stable connections for these shorter interactions. However, existing protocols in this model do not match the level of efficiency that is obtained by traditional MPC protocols. In the first part of this thesis, we study Fluid MPC, and in particular, introduce Fluid MPC protocols with efficiency that matches those of traditional MPC.

The second deficiency of MPC which we study in this thesis is that general-purpose protocols often are still not efficient enough to be used in practice. One way to resolve this is by using protocols that are tailor-made for specific applications. One such application that has gained recent attention is called *Private Join and Compute (PJC)*. In this application, two parties come together with input sets and associated values for each item in their sets. The goal is to privately compute a function over the associated values of the intersection of the two sets. In practice, the size of the intersection is quite small, and therefore the private computation of the intersection is actually much more expensive than whatever computation that needs to be done over it. In the second part of this thesis, we improve the efficiency of tailor-made state-of-the-art protocols that are used to privately compute the intersection, thus improving the efficiency of prior PJC protocols.

Contents

List of Figures

List of Tables

1 | Introduction

Secure Multi-Party Computation (MPC) is a field of cryptography that has been studied extensively since the 1980s [Yao 1986; Goldreich et al. 1987; Chaum et al. 1987]. Broadly speaking, MPC allows a set of mutually distrusting parties to *privately* compute a function of their inputs; i.e., leaking nothing beyond the output of the function. Parties in an MPC protocol communicate with one another in order to make progress on computing the desired function of their private inputs, until they obtain the output. MPC has tremendous potential to make various real-world distributed computations private. Indeed MPC has been deployed in pratice for several applications—(i) J.P. Morgan Privacy-Preserving Inventory Matching, in which potential buyers and sellers of various stocks are paired without revealing their orders to anyone else [Polychroniadou et al. 2023]; (ii) Google Private Join and Compute, in which companies and ad providers can measure the success of ad campaigns without revealing sensitive user information [Private Join and Compute 2019]; (iii) Coinbase Wallet as a Service, in which user cryptocurrency transactions are signed in a distributed fashion by the user and several Coinbase servers such that no single server (or even a few) can produce signatures themselves [Wallet as a Service 2023]; and many more (see [MPC Deployments 2023]).

More formally, MPC protocols typically assume that all parties have point-to-point communication channels with one another that are private and authenticated (i.e., anything that is delivered to one party was in fact sent by the other),[1] and they provide varying levels of privacy guarantees.

[1]How to actually achieve this is a more fundamental (but non-trivial) question of cryptography [Diffie and Hellman 1976].

Privacy in MPC must hold even if an unknown subset of the parties *up to a certain size* act together to try to undermine the privacy of the other (honest) parties. We call such algorithms controlling the collusion of a subset of corrupt parties trying to learn the private information of honest parties the *adversary*. Most generally, such an adversary may act arbitrarily on behalf of the corrupt parties to try to learn private information of the honest parties; in particular, the corrupt parties may not follow the specified steps of the MPC protocol in which they act. We call such an adversary *active*. Some protocols also consider a *passive* adversary, which requires that even corrupt parties follow the protocol specification, while the adversary tries to break the privacy of honest parties. Such protocols of course are not considered as secure as those that protect against active adversaries, yet are more efficient. We will focus on active adversaries in this thesis.

As mentioned above, MPC protocols are specifically designed to tolerate corruption subsets of the parties up to a certain size:

- In the *dishonest majority* setting, MPC protocols are able to provide security for honest parties even if an arbitrary number of other parties are corrupted. While the dishonest majority setting provides the greatest resilience in terms of number of corrupted parties, it is known that protocols in this setting can only be secure against computationally-*bounded* adversaries[2] and moreover require computational assumptions for security [Rabin and Ben-Or 1989]. These protocols are also generally less efficient than protocols in the other settings. However, protocols in this setting can have a circuit- and input-independent *preprocessing* phase (which can be performed at any time before the actual function computation) that is secure against only computationally-bounded adversaries and relies on computational assumptions, but then an *online* phase which is *statistically-secure*—that is, *unconditionally* secure against computationally-*unbounded* adversaries except with some very small probability that is inverse-super-polynomial in some *security parameter* λ.[3] The online phase is

[2]I.e., those that run in time poly(λ), where λ is some *security parameter*.

[3]I.e., protocols in this setting may be insecure with probability at most $\lambda^{-\omega(1)}$.

thus typically more efficient than the preprocessing phase.

A special case of MPC in the dishonest majority setting is Secure *Two-Party* Computation (2PC), in which only one of the parties may be corrupted. We will focus on this setting in the second part of the thesis.

- In the *honest majority* setting, MPC protocols are able to provide security for honest parties as long as if less than half of the parties are corrupted. Protocols in this setting are more efficient than the dishonest majority and can be statistically-secure. However, protocols in this setting cannot be *perfectly-secure*—that is, unconditionally secure against computationally-unbounded adversaries with probability 1 [Lamport et al. 1982].

- In the *two-thirds honest majority* setting, MPC protocols are able to provide security for honest parties as long less than a third of the parties are corrupted. Protocols in this setting typically are more efficient than those in the honest majority settings and can be perfectly secure.

Throughout this thesis, we will only consider protocols in these three settings achieving the best possible security guarantee—i.e., computational, statistical and perfect security, respectively.

It is easy to see that an active adversary may somehow disrupt the execution of MPC protocols (e.g., by not sending messages for corrupt parties). In this case, the honest parties may not be able to actually obtain outputs of the function being computed. Indeed, there are different types of *output guarantees* which MPC protocols can achieve:

- The strongest guarantee is *guaranteed output delivery* or G.O.D. for short. In this case, despite arbitrary behavior from the adversary, the honest parties always obtain the output of the function. Unfortunately, G.O.D. cannot always be achieved—in fact, it is impossible for dishonest majority protocols to achieve it [Cleve 1986]. However, both honest and two-thids honest majority protocols can achieve it.

- The next strongest guarantee is called *fairness*. In protocols with fairness, corrupt parties obtain the output *only if* honest parties do too. However, it may still be the case that no one gets output. It is still impossible for dishonest majority protocols to achieve fairness [Cleve 1986]! Yet, the notion is still meaningful for honest and two-thirds honest majority protocols despite the ability to achieve G.O.D. in these settings—protocols with fairness are generally more efficient than those with G.O.D.

- The weakest guarantee that can be provided is *security with abort*. In this case, the protocol execution may abort at any time. However, it may still be the case that the adversary learns the output of the function, even if none of the honest parties learn it. Dishonest majority protocols can indeed achieve security with abort. But, again, this notion is still meaningful for the honest and two-thirds honest majority settings, since protocols that achieve security with abort are more efficient than those with fairness and G.O.D.

Since this thesis focuses on *efficiency*, we will only construct protocols that achieve security with abort.

Theoretical feasibility results for MPC in all settings have been known since the inception of the field (e.g., [Ben-Or et al. 1988; Chaum et al. 1987] for perfect security with G.O.D., [Rabin and Ben-Or 1989] for statistical security with G.O.D., and [Yao 1986; Goldreich et al. 1987] for computational security with abort). In the meantime, protocols for MPC have seen great improvements in efficiency, through works including [Damgård et al. 2012; Ben-Efraim et al. 2019] for dishonest majority, [Damgård and Nielsen 2007; Genkin et al. 2014; Chida et al. 2018; Goyal and Song 2020; Boyle et al. 2020; Goyal et al. 2021a; Escudero et al. 2022] for honest majority, and [Beerliová-Trubíniová and Hirt 2008; Abraham et al. 2023] for two-thirds honest majority. In fact, MPC is now deployed in the real world in various settings (see [MPC Deployments 2023]).

However, MPC still only enjoys a somewhat limited set of applications in practice. This is because even though the efficiency of MPC protocols has vastly improved, MPC in general does have some shortcomings. First, MPC protocols are inherently distributed protocols that must utilize communication over some network, such as the internet, and thus must make some assumptions about this network. On one hand, real-world networks can be unstable, nodes can drop in and out of networks via crashes/disconnections, messages can take different amounts of time to be sent from one node to another, etc. Yet, on the other hand, most MPC protocols are specifically designed to interact over stable networks. One crucial consequence of this is that if a party's device crashes or a message from them does not arrive on time, then this party is treated as corrupted. This is bad for several reasons. One, protocols only have a limited budget of corrupted parties they can tolerate, before privacy of honest parties can be compromised. Therefore, using up this budget for honest parties who may just crash, etc, is unsatisfactory. Two, once a party is considered corrupt, privacy is no longer provided for them, from a definitional standpoint. Therefore, honest parties who crash, etc, may have their privacy violated.

There has been a line of works that attempt to address this issue with MPC [Choudhuri et al. 2021; Gentry et al. 2021b; Badrinarayanan et al. 2020; Damgård et al. 2021; Guo et al. 2019; Rachuri and Scholl 2022]. In the first part of this thesis, we focus on MPC protocols in the so-called *Fluid* model [Choudhuri et al. 2021], upon which we will elaborate below.

Another shortcoming of (general-purpose) MPC is that protocols aim to compute *any* given function and as a result, despite the great improvements in efficiency over the past few decades, there are some tasks for which (general-purpose) MPC protocols are just too slow or expensive in terms of communication. Yet, in practice many use cases must of course be low-latency to be viable for use. Moreover, computation and communication costs must not be prohibitively high in order to enable adoption by participants in the real-world. Fortunately, many MPC applications can be precisely defined, and there has been a push in recent years to craft tailor-made protocols for these applications, instead of always relying on general-purpose MPC protocols. For example, in the second part of this thesis, we focus on protocols for *Private Join and Compute* [Private Join and Compute 2019], upon which we will elaborate below.

1.1 Fluid Secure Multi-Party Computation

We now discuss the *Fluid* MPC model, introduced by [Choudhuri et al. 2021]. This model reduces the effects of unstable networks on parties by not requring them to be online and participating in the *entire* protocol. Instead, parties may come and go, more or less as they please, participating in the protocol for as little as just one communication round.[4] More specifically, parties from some global pool form *committees* which are subsets of the pool and come online to participate in the protocol for a given number of rounds. Once such a committee reaches their final round, they afterwards *transfer* the state of the computation to the next committee, who then continues the computation. Importantly, the parties in one committee only know the identities of the parties in the immediately following committee, and not those of any future committees.

The original work of [Choudhuri et al. 2021] studies Fluid MPC with *maximal fluidity*. With maximal fluidity, each committee only comes online for a *single* round of interaction. That is, each committee receives the state of the computation from the previous committee, performs some local computation, and then forwards the new state to the next committee. Their work provides a protocol that achieves *statistical security with abort* in the setting in which *each* committee has an *honest majority* of parties. Later, [Rachuri and Scholl 2022] provide a protocol for Fluid MPC with maximal fluidity that has an online phase that achieves *statistical security with abort* in the setting in which each committee has a *dishonest majority* of parties. Since each committee has a dishonest majority of parties and the online phase is statistically-secure, this protocol requires a preprocessing phase to be performed by all of the N members of the global pool of parties that may participate in committees during the online phase of the protocol. This pool is much bigger than the n members that participate in each committee (i.e., $N \gg n$). Note that importantly, this is because the identity of the committees is not necessarily decided ahead of time, before the

[4]A communication round is defined as a set of messages first being sent (simulatenously) from some of the participants in one direction along network channels, and then being delivered to their recipients.

preprocessing phase. More recently, [David et al. 2023] provide a protocol for Fluid MPC with maximal fluidity that achieves *G.O.D.* in the setting in which each committee has a *two-thirds honest majority* of parties.

All three of these works make great strides towards efficient Fluid MPC protocols that can be used in practice. However, they all do not realize the same efficiency as their counterparts in the *traditional* MPC setting. Traditional MPC protocols for all three corruption settings we have discussed, with maximal security and output guarantees, exist with communication complexity *linear* in the number of participants (such as [Damgård et al. 2012; Ben-Efraim et al. 2019] for dishonest majority, [Damgård and Nielsen 2007; Genkin et al. 2014; Chida et al. 2018; Goyal and Song 2020; Boyle et al. 2020; Goyal et al. 2021a; Escudero et al. 2022] for honest majority, and [Beerliová-Trubíniová and Hirt 2008; Abraham et al. 2023] for two-thirds honest majority). That is, as the number of participants grows, the communication per party stays constant in these protocols. Unfortunately, the three aforementioned works do not achieve the analogue for (maximally) Fluid MPC. For [Choudhuri et al. 2021] and [Rachuri and Scholl 2022], the protocols achieve communication complexity *quadratic* in the size of each committee. That is, as the size of each committee grows, the communication per party grows proportionally. For [David et al. 2023], the protocol achieves communication complexity that grows with n^9, where n is the size of each committee (note that they achieve the strongest output guarantee—G.O.D.). Thus, there is an unsatisfactory gap between the asymptotic communication complexity of traditional MPC prtocols and those in the (maximally) Fluid setting. Indeed, the communication complexity of existing Fluid MPC protocols may be prohibitively high for scaling to large number of parties.

1.1.1 Our Results

In this thesis, we close the gap in communication complexity between traditional MPC protocols and Fluid MPC protocols with maximal fluidity in all three corruption settings. Therefore, we achieve MPC that can handle some instability of the underlying communication network (by not requiring all parties to have reliable access to it for the entire protocol), while still achieiving great efficiency. Our protocols work over *layered* arithmetic circuits over a finite field $\mathbb{F}$ with $|C|$ gates. A layered circuit is composed of input, addition, multiplication, and identity gates, where all output wires of a given layer go only to the immediately next layer. We let w_ℓ be the width of the ℓ-th layer of C and for some gate g in C, we let $\ell(g)$ be the index of the layer that g belongs to in C. As with many traditional MPC protocols that achieve linear communication complexity [Damgård et al. 2010; Damgård and Nielsen 2007], we assume that the width of all layers of considered circuits is at least proportional to the committee size, n. We remark, however, that even if this is not the case, our protocols are still a strict improvement over the prior works.

In the dishonest majority setting, we provide a Fluid MPC protocol with maximal fluidity that achieves security with abort and has communication complexity $O(n|C|)$ during the statistically-secure online phase, where n is the size of each committee (assuming each committee is the same size). This protocol works given some global preprocessing among the whole global pool of parties, without knowing the future assignment of parties to committees:

Theorem 1.1 (Informal [Bienstock et al. 2023a]). *For a layered arithmetic circuit C over a finite field $\mathbb{F}$, there exists a statistically-secure fluid MPC protocol with maximal fluiditiy in the preprocessing model which securely computes C, with abort, in the presence of an active adversary controlling up to $t \geq n/2$ parties, where n is the size of each committee. The amount of preprocessed data used per party is $\Omega(N \cdot |C|)$, where N is the number of parties in the global pool. The communication cost per gate g is $O(n^2/w_{\ell(g)})$. In particular, if the width of all layers is $w = \Omega(n)$, then the total cost is $O(n|C|)$ elements of communication.*

The amount of preprocessed data per party, $\Omega(N \cdot |C|)$, may indeed seem prohibitively large for practice. Unfortunately, we show that for linear communication complexity, such a large amount of preprocessing is necessary. More specifically, we show that the amount of preprocessing per party required in the statistically-secure dishonest majority setting to securely transfer the protocol execution state st from one committee to the next (which is an essential building block of Fluid MPC) is $\Omega(N \cdot |\mathrm{st}|)$:

Theorem 1.2 (Informal [Bienstock et al. 2023a])**.** *A secure message transmission protocol for messages of length λ with two n-party committees that uses $o(n^2 \cdot \lambda)$ total communication must have $\Omega(N \cdot \lambda)$ preprocessed data.*

In particular, if each committee computes the output of at most one circuit layer at a time, and each party may participate in a constant fraction of committees in the worst-case, then the total preprocessing per party must be $\Omega(N \cdot |C|)$. Note that such large preprocessing for secure state transfer is not needed if quadratic communication complexity can be tolerated, as then committees can just use a trivial *resharing* protocol [Ben-Or et al. 1988]. However, the existing Fluid *MPC* protocol with $O(n^2|C|)$ communication in this setting [Rachuri and Scholl 2022] still requires $\Omega(N|C|)$ preprocessing per party.[5]

In the honest majority setting, we provide a Fluid MPC protocol with maximal fluidity that achieves security with abort and has communication complexity $O(n|C|)$. This protocol requires no preprocessing among the parties and is statistically-secure:

Theorem 1.3 (Informal [Bienstock et al. 2023a])**.** *For a layered arithmetic circuit C over a finite field $\mathbb{F}$, there exists a statistically-secure fluid MPC protocol with maximal fluiditiy which securely computes C, with abort, in the presence of an active adversary controlling up to $t < n/2$ parties, where n is the size of each committee. The communication cost per gate g is $O(n^2/w_{\ell(g)})$. In particular, if the width of all layers is $w = \Omega(n)$, then the total cost is $O(n|C|)$ elements of communication.*

Finally, in the two-thirds honest majority setting, we provide a Fluid MPC protocol with maximal fluidity that achieves security with abort and has communication complexity $O(n|C|)$. This protocol requires no preprocessing among the parties and is perfectly-secure:

[5]This is due to generating preprocessed correlations between parties which enables secure computation of multiplication gates in the online phase.

Theorem 1.4 (Informal [Bienstock et al. 2023b]). *For a layered arithmetic circuit C over a finite field $\mathbb{F}$, there exists a perfectly-secure fluid MPC protocol with maximal fluiditiy which securely computes C, with abort, in the presence of an active adversary controlling up to $t < n/3$ parties, where n is the size of each committee. The communication cost per gate g is $O(n^2/w_{\ell(g)})$. In particular, if the width of all layers is $w = \Omega(n)$, then the total cost is $O(n|C|)$ elements of communication.*

1.1.2 Related Work

Our work expands upon the work of [Choudhuri et al. 2021; Rachuri and Scholl 2022; David et al. 2023] in the Fluid MPC setting. However, in the broader direction of MPC in unstable networks, several other references exist. We survey them here.

Fail-stop adversaries. A series of works have studied the setting of MPC, where the adversary is allowed to not only corrupt some parties passively/actively, but also cause some parties to fail (e.g. [Fitzi et al. 1998] and subsequent works). This can be seen as similar to the Fluid setting, where parties who participate in one committee may never participate again in another committee. However, one main difference is that unlike in the committee approach of Fluid, the set of parties that fail and thus exit the computation are not known to the rest of the parties. Second, and most crucially, once a party is set to fail by the adversary, it does not return to the computation, whereas parties in Fluid can arbitrarily be placed in several non-consecutive committees.

LazyMPC. The work of [Badrinarayanan et al. 2020] considers an adversary that can set parties to be offline in any round (called "honest but lazy" in that work). This work differs from ours in several places. First, the authors focus only on the case of computational security, making use of rather strong techniques such as multi-key fully homomorphic encryption. Second, the parties that are chosen to be "lazy" are not known to the other parties. Third, once a party becomes offline, or "lazy", in their model it is assumed not to come back.

Synchronous but with partition tolerance. Recently, the work of [Guo et al. 2019] designed MPC protocol in the so-called "sleepy model", which enables some of the parties to lag behind the protocol execution, while not being marked as corrupt. This could be achieved with an asynchronous protocol, naturally, but the main result of [Guo et al. 2019] is obtaining such protocols without the strong threshold assumptions required to obtain asynchronous protocols. In particular, the authors obtain computationally secure constant-round protocols, assuming that the set of "fast"-and-honest parties in every round constitutes as majority, an assumption that is shown to be necessary.

Phoenix. The work of [Damgård et al. 2021] proposes a model that is similar to the one in [Guo et al. 2019] in that parties can go offline for short periods of time, but unlike [Guo et al. 2019], the parties are not assumed to receive messages while they are offline ([Guo et al. 2019] considers unstable parties as "slow", meaning they still receive messages but they might not do so on time; in contrast, [Damgård et al. 2021] considers these parties to be potentially entirely offline). The work proposes solutions in their "Phoenix" model for MPC with perfect, statistical and computational security, and prove exact conditions on the adversary under which these are possible.

YOSO. In the recent work of Gentry et al. [Gentry et al. 2021a], the "You Only Speak Once" model for MPC is introduced. In this model, the basic assumption is that the adversary is able to take a party down as soon as that party sends a message - using, say, a denial of service attack. Although some number of parties are assumed to be alive and can receive messages, no particular party is guaranteed to come back (which is the major difference to our model). Instead, the YOSO model breaks the computation into small atomic pieces called *roles* where a role can be executed by sending only one message. The responsibility of executing each role is assigned to a physical party in a randomized fashion. The assumption is that this will prevent the adversary from targeting the relevant party until it sends its (single) message. This means that one should think of the entire set of parties as one "community" which as a whole is able to provide secure computation as a service. In a sense, YOSO aims to make progress and keep the computation alive without any guarantees for particular physical parties such as contributing inputs and receiving the output. This makes good sense in the context of a blockchain, for instance. On the other hand, the demand that the MPC protocol must be broken down into roles makes protocol design considerably harder, particularly for information theoretically secure protocols. An additional caveat with the YOSO model is that one can only have information theoretically or statistically secure protocols assuming that the role assignment mechanism is given as an ideal functionality, and an implementation of such a mechanism must inherently be only computationally secure. In comparison, our model assumes a somewhat less powerful adversary who must allow a physical party to come back after being offline. This allows for much easier protocol design, information theoretic security based only on point-to-point secure channels, and allows termination such that all parties can provide input and get output.

1.2 Private Information Retrieval and its Application to Private Join and Compute

Some functionalities can be prohibitively expensive to compute using general-purpose MPC protocols. For these functionalities, it is better to design tailor-made protocols which are designed with the specific task in mind, to get better efficiency. One functionality which is particularly important in practice is called *Private Join and Compute (PJC)* [Private Join and Compute 2019]. In general, PJC is a two-party functionality in which two parties input sets X and Y (possibly with associated data for each set element), and then the functionality computes $X \cap Y$ and outputs secret shares to the two parties of the elements in $X \cap Y$ (along with the associated data, if any). There are more specific variants of PJC, in which some computation is also performed on the elements of $X \cap Y$, before the output is sent to one or both of the parties. For example, in Inner Product PJC [Lepoint et al. 2021], each element x_i of X for $i \in [\ell]$ has some associated data w_i

and each element y_j of Y for $j \in [n]]$ has some associated data v_i, then the functionality computes and outputs to one or both of the parties $\sum_{i\in[\ell],j\in[n],x_i=y_j} w_i v_j$. Note that typically $\ell \ll n$. Usually for PJC, the cost of the *compute* part of the protocol is determined by the size of the intersection, which is often quite small, and thus the dominant costs come from determining the intersection. Thus, it is important to design constructions specifically-built for the intersection part, in order to maximize the efficiency.

There are several applications of Inner Product PJC. One such application is for privately computing the effectiveness of advertising campains [Ion et al. 2020; Lepoint et al. 2021]. In this application one party, typically some company selling products, has a database of parties who have bought some products with associated values of how much money they spent, and the other party is an ad supplier, which has a database of parties who have seen some ads, potentially with associated values that can be e.g., how long they spent watching an ad, or just associated values equal to 1. Inner Product PJC can thus be used to privately compute aggregate statistics such as how much users who saw an ad spent in total on the company's products.

A primary building block of PJC protocols, which is used to compute the intersection $X \cap Y$ of the two parties' sets, is *batch Private Information Retrieval (batch PIR)* [Lepoint et al. 2021]. To define batch PIR, we start with defining *single-query* PIR. In single-query PIR, there is a server with a database of n items and a client who wants to retrieve the i-th item of the database. For security, the server should not learn anything about the index, i, for which the client queries. In batch PIR, the client asks for a set I of size $|I| = \ell \ll n$ of items of the database, and the server should not learn anything about the set I. There is also a keyword (batch) PIR variant, in which the server holds a database of key-value pairs (where the key space is sparse) and the client's queries are keys. Batch PIR protocols can be easily adapted so that the client and server get secret shares of those items of I that are in the database, and thus are used as a building block for computing the intersection $X \cap Y$ in PJC protocols [Lepoint et al. 2021]. There are also versions of PIR in which two servers hold the database and independently compute responses for the client, which the client uses together to determine the answer. These protocols have the benefit of being more efficient, while requiring the strong assumption that the two servers do not collude with each other to break the security of the client.

Other Applications of PIR. PIR and its variants also enjoy several other applications beyond PJC. Some examples include anonymously retrieving encrypted messages from a central delivery server [Angel and Setty 2016], privately fetching advertisements relevant to user interests, and more. Furthermore, using the classical composition with Oblivious Pseudo-Random Functions (OPRFs) due to [Freedman et al. 2005], PIR can be used to build (unbalanced) labeled Private Set Intersection (PSI). Labeled PSI is actually the same as PIR with the exception that also all server database items that are not in the intersection should remain private from the client (which is not required in PIR). Unbalanced Labeled PSI is particularly for the case in which the client's query set is much smaller than the server's database (as in typical in PIR). Labeled PSI can then be used for several applications. For example, Labeled PSI can be used to build contact discovery [Contact Discovery 2017] protocols in which a user registers for some service, and privately queries the server for its contacts (via e.g., phone numbers) who also use the service, and their corresponding profiles. Labeled PSI can also be used for password leak checks [Password Checkup 2019; Password Monitor 2021] in which a user creating a username and password for some service privately checks their password against a database of passwords (stored on the server) known to be leaked on the dark web. There are many other applications of PIR and PSI besides those mentioned above.

1.2.1 Our Results

In this thesis we improve the communication complexity of state-of-the-art batch PIR protocols, thus improving the communication complexity of the main step of state-of-the-art PJC and labeled PSI protocols. Recall that in a single query of batch PIR, the client asks for a set of ℓ entries from the database of size $n \gg \ell$. The naive approach to solve this problem would be to simply

	Encoding Size	Encoding Time	Decoding Time
Choi *et al.* [Choi et al. 2021]	$O(\ell\lambda)$	$O(n\lambda)$	$O(\ell\lambda)$
Liu and Tromer [Liu and Tromer 2022]	$O(\ell \log^2 \ell \log \lambda)$	$O(n\ell)$	$O(\ell^3)$
Fleischhacker *et al.* [Fleischhacker et al. 2023]	$O(\ell)$	$O(n \log n)$	$O(\ell\sqrt{n})$
Fleischhacker *et al.* [Fleischhacker et al. 2023]	$O(\ell\lambda)$	$O(n\lambda)$	$O(\ell\lambda)$
Ours: LSObvCompress	$(1+\epsilon)\ell$	$O(n\lambda)$	$O(\ell\lambda)$

Table 1.1: Comparison of ciphertext compression for n ciphertexts with ℓ non-zero values for failure probability at most $2^{-\lambda}$. Encoding size is measured in number of ciphertexts.

run ℓ independent queries of a single-query PIR protocol. However, this incurs a high total computational overhead of $\Omega(\ell n)$ for the server, as single-query PIR protocols must incur $\Omega(n)$ server computational complexity for answering each query [Beimel et al. 2004]. Instead, state-of-the-art batch PIR protocols opt to reduce the total server computational overhead to $O(n)$, while slightly increasing the communicaction complexity. Indeed, [Angel et al. 2018] presented a solution that reduces server computation to $3n$ while requiring 1.5ℓ independent queries of a single-query PIR protocol on smaller databases. Thus, unfortunately, the size of requests and responses for the batch PIR are 50% larger than the naive approach of ℓ single-query PIR executions on the server's database of size n.

Single-query PIR schemes typically work by encrypting some item query in a ciphertext as the request sent to the server, and then using this ciphertext on the server side to (homomorphically) compute a ciphertext containing the corresponding database entry, which is sent to the client as the response. While prior works attempted to reduce communication by packing multiple single-query PIR requests into a single ciphertext [Angel et al. 2018; Ali et al. 2021], as well as packing multiple responses (for small database entries) into a single ciphertext [Mughees and Ren 2023], these solutions still explicitly require 0.5ℓ single-query PIR requests and responses, which the client ultimately discards as garbage. Moreover, packing responses only works for small database entries. In this chapter, we present two novel compression techniques that avoid the unecessary communication that arises from these 0.5ℓ *dummy* queries. As a result, we get improved state-of-the-art single- and two-server batch PIR, PJC, and labeled PSI protocols.

Oblivious Ciphertext Compression. The first compression problem that we study we call *oblivious ciphertext compression*. In this problem, there is a client and a server. The server has n (ordered) ciphertexts encrypted under the client's key, of which $\ell < n$ encrypt non-zero values (the rest encrpypt zero). Moreover, the client knows the locations of the ℓ ciphertexts that encrypt non-zero values, while these locations are unknown to the server. The goal is to enable the server to compress the n ciphertexts to as close to ℓ ciphertexts as possible, before sending them to the client, who can then correctly decode the n ciphertexts.

We present a scheme for this problem that produces encodings as small as 1.05ℓ ciphertexts, while only requiring homomorphic addition of ciphertexts. Furthermore, the server only needs to do $O(n\lambda)$ homomorphic additions and the client only needs to perform $O(\ell\lambda)$ plaintext additions:

Theorem 1.5 (Informal [Bienstock et al. 2024])**.** *There exists an* oblivious ciphertext compression *protocol for compressing some n additively-homomrphic ciphertexts such that $\ell < n$ plaintexts are non-zero, which outputs $(1 + \epsilon)\ell$ ciphertexts, for any $\epsilon > 0$, requires $O(n\lambda)$ homomorphic additions by the server and $O(\ell\lambda)$ plaintext additions by the cilent.*

Our scheme uses novel techniques to compress the ciphertexts by encoding them with random linear systems that are efficiently solvable.

Our scheme significantly outperforms any prior compression schemes that can be used in our setting. In particular, all prior schemes with efficient encoding and decoding compress to $O(\ell\lambda)$ ciphertexts, which is much larger than 1.05ℓ in practice. Only one previous solution produces encodings with $O(\ell)$ ciphertexts [Fleischhacker et al. 2023], but the decoding requires computing $O(\ell\sqrt{n})$ discrete logarithms, which is far too expensive. See Table 1.1 for more comparisons. We remark, however, that prior works study a more challenging version of this problem (see Section 1.2.2).

Oblivious Ciphertext Decompression. The second compression problem that we study we call *oblivious ciphertext decompression*. In this problem there is also a client and a server. However,

	Client Storage	Request Overhead	Response Overhead
Baseline	$O(1)$	1x	1x
Cuckoo Hashing [Angel et al. 2018]	$O(n)$	$\lceil 1.5/r \rceil$x	1.5x
Vectorized [Mughees and Ren 2023]	$O(n)$	$\lceil 1.5/r \rceil$x	$\lceil 1.5/d \rceil$x
Keyword [Patel et al. 2023]	$O(1)$	$\lceil 1.5/r \rceil$x	1.5x
Distributed Point Function (DPF)* [Boyle et al. 2016]	$O(1)$	1.5x	1.5x
Ours: Single-Server	$O(1)$	$\lceil (1+\epsilon)/r \rceil$x	1.5x
Ours: Single-Server	$O(1)$	$\lceil 1.5/r \rceil$x	$\lceil (1+\epsilon)/d \rceil$x
Ours: Two-Server*	$O(1)$	1.5x	$(1+\epsilon)$x

Table 1.2: Keyword batch PIR comparisons for retrieving ℓ entries from n-entry database. Request and response overhead is compared to baseline of performing ℓ independent single-query PIR executions. We use r and d to denote the number of requests and plaintext database entries that can fit into a single ciphertext. Asterisks(*) denote two-server PIR protocols.

this time the client has n (ordered) plaintext values, of which $\ell < n$ are non-dummy values (the rest are dummy values). The client must compress and encrypt the n plaintext values into as close to ℓ ciphertexts as possible, before sending them to the server, who can then correctly decompress these ciphertexts back into n new ciphertexts such that for the ℓ locations that originally had non-dummy plaintext values, the corresponding ciphertexts encrypt these values. Moreover, the server must perform this decompression without knowledge of the ℓ non-dummy locations, and must not learn anything about these ℓ non-dummy locations nor their values.

We present a scheme for this problem that produces encodings as small as 1.05ℓ ciphertexts, while only requiring homomorphic addition of ciphertexts. Furthermore, the server only needs to do $O(n\lambda)$ homomorphic additions, and the client only needs to perform $O(\ell\lambda)$ plaintext additions:

Theorem 1.6 (Informal [Bienstock et al. 2024])**.** *There exists an* oblivious ciphertext decompression *protocol for compressing some n plaintexts such that $\ell < n$ plaintexts are non-dummy, which outputs $(1+\epsilon)\ell$ additively-homomorphic ciphertexts, for any $\epsilon > 0$, requires $O(n\lambda)$ homomorphic additions by the server and $O(\ell\lambda)$ plaintext additions by the cilent.*

To our knowledge, no prior works are applicable to this specific problem.

Batch PIR. We apply our compression techniques to state-of-the-art batch PIR schemes to achieve new schemes with reduced communication in both the single- and two-server settings. Our techniques work for both small and large database entries. See Table 1.2 for detailed comparisons to prior work.

The framework of [Angel et al. 2018] introduces 0.5ℓ *dummy* queries of a single-query PIR protocol, as part of the overall batch PIR query. This therefore results in 0.5ℓ *dummy* responses of a single-query PIR protocol, as part of the overall batch PIR response. If r single-query PIR requests fit into a single ciphertext (resp. d database entries fit into a single ciphertext), then this is $0.5\ell/r$ ciphertexts that contain dummy requests (resp. $0.5\ell/d$ ciphertexts that contain dummy reponses). We note that state-of-the-art batch PIR protocols use underlying single-query PIR protocols which work with additively-homomorphic encryption schemes (in fact, somewhat-homomorphic). Thus, we can apply our compression schemes to these protocols.

First, we apply our oblivious ciphertext compression to obtain batch PIR with response size as small as $1.05\ell/d$, regardless of database entry size. Indeed, since ℓ of the batch PIR responses encrypt non-zero values, whose location the client knows, and the remaining 0.5ℓ encrypt zero values,[6] we can apply our oblivious ciphertext compression scheme:

Theorem 1.7 (Informal [Bienstock et al. 2024])**.** *There exists a* batch PIR *protocol for databases of n b-bit entries and batch size ℓ, that has query size proportional to $1.5\ell/r$, response size proportional to $(1+\epsilon)\ell/d \cdot b$, for any $\epsilon > 0$, and server computation $O(n)$.*

Next, we apply our oblivious ciphertext decompression to obtain batch PIR with request size as small as 1.05ℓ, regardless of database entry size. Indeed, since ℓ of the batch PIR requests are non-dummy, whose location the client knows, and the remaining 0.5ℓ are dummy, we can apply our oblivious ciphertext decompression scheme:

Theorem 1.8 (Informal [Bienstock et al. 2024])**.** *There exists a* batch PIR *protocol for databases of*

[6]We can easily enforce this.

n b-bit entries and batch size ℓ, that has query size proportional to $(1+\epsilon)\ell/r$, for any $\epsilon > 0$, response size proportional to $1.5\ell/d \cdot b$, and server computation $O(n)$.

While theoretically we can apply both oblivious ciphertext compression and decompression simultaneously to state-of-the-art batch PIR protocols, we show that this does not lead to any concrete gains in practice. Indeed, the noise that results from applying the extra operations of these two schemes to the underlying somewhat-homomorphic encryption scheme of the batch PIR protocols is such that the ciphertexts need to be too large (see Section 4.4.3).

Finally, we also show that similar response reduction may be obtained in state-of-the-art two-server batch PIR protocols, by applying our oblivious ciphertext compresssion scheme to them.

As a result of these improvements to batch PIR, we conjecture that we can obtain concrete efficiency improvements when applied to PJC. We also see concrete improvements to labeled PSI. In particular, if one combines our state-of-the-art batch PIR with oblivious ciphertext compression and a state-of-the-art OPRF scheme, we obtain a labeled PSI protocol [Freedman et al. 2005] with a 65-88% reduction in communication and comparable computation to prior solutions [Chen et al. 2018; Cong et al. 2021].

1.2.2 Related Work

Ciphertext Compression. Variants of ciphertext compression have been studied in the past. Liu and Tromer [Liu and Tromer 2022] implicitly studied oblivious ciphertext compression without explicitly defining the primitive. In their scheme, they use sparse linear random codes that result in larger encodings and slower decoding time (see Figure 1.1), and, if instantiated with a FHE scheme, larger parameters for that scheme. Angel *et al.* [Angel et al. 2018] used packing and vectorization techniques to reduce request communication in PIR. Mughees and Ren [Mughees and Ren 2023] also showed vectorization techniques may be used to reduce response communication in batch

PIR. Fleischhacker *et al.* [Fleischhacker et al. 2023] studied a more challenging variant of our setting where neither the decompressor (client) nor the compressor (server) know the identity of the the non-zero ciphertexts. As a result, their schemes have worse compression rate and more expensive compression and decompression algorithms. The same problem was implicitly studied in [Choi et al. 2021].

PIR. Single-server PIR was first studied by Kushilevitz and Ostrovsky [Kushilevitz and Ostrovsky 1997]. Follow-up works constructed PIR from various other assumptions [Cachin et al. 1999; Paillier 1999; Damgård and Jurik 2001; Lipmaa 2005; Gentry and Ramzan 2005]. More recent works have studied concretely efficient protocols from lattice-based homomorphic encryption [Aguilar Melchor et al. 2016; Angel and Setty 2016; Angel et al. 2018; Gentry and Halevi 2019; Park and Tibouchi 2020; Ali et al. 2021; Mughees et al. 2021; Ahmad et al. 2021; Menon and Wu 2022; Mahdavi and Kerschbaum 2022; Patel et al. 2023].

PIR has also been studied in the setting of multiple, non-colluding servers. A line of work has studied the communication efficiency with information-theoretic security (see [Chor et al. 1998; Efremenko 2009; Dvir and Gopi 2015] and references therein). Recent works have studied concretely efficient two-server PIR with computational security using distributed point functions [Gilboa and Ishai 2014; Boyle et al. 2016; Hafiz and Henry 2019].

Batch PIR. Batch PIR has been studied heavily in the past. Beimel *et al.* [Beimel et al. 2000] presented a method to reduce server computation using matrix multiplication. Groth *et al.* [Groth et al. 2010] presented a communication-optimal scheme adapting the scheme in [Gentry and Ramzan 2005]. Another line of work (see [Ishai et al. 2004; Lueks and Goldberg 2015; Henry 2016; Yeo 2023] and references therein) presented batch codes that transforms any single-query PIR into a batch PIR. More recent work [Angel and Setty 2016; Angel et al. 2018] introduced probabilistic batch codes that result in the most concretely-efficient batch PIR schemes to date. Mughees and Ren [Mughees and Ren 2023] introduced vectorization techniques to reduce server responses for small database entries. Patel *et al.* [Patel et al. 2023] presented keyword PIR schemes that can remove the client mapping.

Labeled PSI. Labeled PSI is a variant where each identifier has an associated data label that should be retrieved. Labeled PSI is most often studied in the unbalanced setting where the receiver's set is much smaller than the sender's set. Many recent works [Chen et al. 2017, 2018; Demmler et al. 2018; Kales et al. 2019; Cong et al. 2021] studied labeled PSI with sub-linear communication in the larger set. The same setting where the receiver only queries for a single item has been studied as symmetric PIR [Gertner et al. 1998; Ali et al. 2021].

2 Preliminaries

2.1 Notation

We will use λ as the security parameter throughout. We will use $x \leftarrow y$ for assignment of variable x to y. Sometimes we will assign a variable x to the output of some algorithm A, denoted, $x \leftarrow A(\cdot)$. We use $x \leftarrow_{\$} \mathcal{X}$ to denote sampling x randomly from distribution $\mathcal{X}$. Sometimes we will sample x based on a randomized algorithm A, denoted, $x \leftarrow_{\$} A(\cdot)$. If A is run on explicit randomness r, we will write $x \leftarrow A(\cdot; r)$.

Linear algebra. We denote column vectors as $\mathbf{v}$ and row vectors as $\mathbf{v}^T$. We denote the i-th entry of $\mathbf{v}$ by $\mathbf{v}_i$. For two vectors n-length vectors $\mathbf{v}$ and $\mathbf{u}$, we denote the dot product operator as $\mathbf{v} \cdot \mathbf{u} = \sum_{i=1}^{n} \mathbf{v}_\mathbf{i} \cdot \mathbf{u}_\mathbf{i}$. We define a $n \times m$ matrix using its column vectors as $\mathbf{M} = (\mathbf{v}_1, \ldots, \mathbf{v}_m)$ where the i-th column vector is $\mathbf{v}_i$ of length n. We may also define a matrix using its row vectors as $\mathbf{M} = (\mathbf{v}_1^T, \ldots, \mathbf{v}_n^T)$ where $\mathbf{v}_i^T$ is the i-th row vector of length m. We denote the matrix-vector product $\mathbf{M} \cdot \mathbf{u} = (\mathbf{v}_1 \cdot \mathbf{u}, \ldots, \mathbf{v}_n \cdot \mathbf{u})$ where $\mathbf{u}$ is a m-length vector. We solve the linear system associated with $n \times m$ matrix $\mathbf{M}$ and n-length vector $\mathbf{u}$ by computing m-length vector $\mathbf{v}$ such that $\mathbf{M} \cdot \mathbf{v} = \mathbf{u}$.

For a vector $\mathbf{v}$ of length n and subset $I = \{i_1, \ldots, i_k\} \subseteq [n]$, we denote by $\mathbf{v}_I = (\mathbf{v}_{i_1}, \ldots, \mathbf{v}_{i_k})$ containing the entries of $\mathbf{v}$ with indices in I. For $n \times m$ matrix $\mathbf{M} = (\mathbf{v}_1^T, \ldots, \mathbf{v}_n^T)$ and subset $I = \{i_1, \ldots, i_k\} \subseteq [n]$, we denote the sub-matrix consisting of row vectors with indices in I as

$\mathbf{M}_{\mathrm{r}(I)} = (\mathbf{v}_{i_1}^T, \ldots, \mathbf{v}_{i_k}^T)$. Similarly, for a $n \times m$ matrix $\mathbf{M} = (\mathbf{v}_1, \ldots, \mathbf{v}_m)$ and subset $I = \{i_1, \ldots, i_k\} \subseteq [m]$, we denote the sub-matrix consisting of column vectors with indices in I as $\mathbf{M}_{\mathrm{c}(I)} = (\mathbf{v}_{i_1}, \ldots, \mathbf{v}_{i_k})$. Additionally, given subset $J \subseteq [n]$, we use $\mathbf{M}_{\mathrm{r}(J)}^{\mathrm{c}(I)}$ to denote the $|J| \times |I|$ matrix consistent of the rows of $\mathbf{M}$ indexed by the set J, and the columns of $\mathbf{M}$ indexed by I.

2.2 Universal hashing.

We make use of universal hash function families in both our honest and dishonest majority protocols. A family of hash functions $\mathsf{H} = \{\mathsf{H}_s : \mathbb{F}_p^T \longrightarrow \mathbb{F}_p\}$ is *universal* if for all $x \neq y \in \mathbb{F}_p^T$,

$$\Pr_s[\mathsf{H}_s(x) = \mathsf{H}_s(y)] \leq 1/p.$$

2.3 Probability and Information Theory

Here we present some additional notation and definitions from probability and information theory. For a random variable X we use $\mathrm{H}(X)$ to represent its *Shannon entropy.* For two random variables X and Y, we define their *mutual information* as:

$$\mathrm{I}(X; Y) = \mathrm{H}(X) - \mathrm{H}(X|Y).$$

Also, for two random variables X and Y over the same space $\mathcal{Z}$, we define the *statistical distance* between their probability distributions as:

$$\mathrm{SD}(X, Y) = \max_{Z \subseteq \mathcal{Z}} |\Pr[X \in Z] - \Pr[Y \in Z]|.$$

Finally, we define the *Kullback-Leibler divergence* of the probability distribution of X from that of Y as:

$$D_{\mathrm{KL}}(X||Y) = \sum_{z \in \mathcal{Z}} \Pr[X = z] \cdot \log\left(\frac{\Pr[X = z]}{\Pr[Y = z]}\right).$$

2.4 Functionalities, Protocols and Procedures

In this work we denote functionalities by $\mathcal{F}$ and some subscript, and protocols by Π and some subscript. We also consider *procedures*, denoted by π and some subscript. These are similar to protocols except that (1) they act like "macros" that can be called within actual protocols and (2) they are not intended to instantiate a given functionality. Instead, security is proven in the protocol where they are used.

3 Secure Multi-Party Computation in the Fluid Model with Linear Communication

In this chapter, we study Fluid MPC. As described in the introduction, Fluid MPC demands less in terms of the stability of communication channels between parties by only requiring them to be online, and thus maintain stable connections, for short amounts of time. We give protocols with linear communication in each of the dishonest majority with preprocessing, honest majority, and two-thirds majority settings. We also prove a lower bound showing that for linear communication in the dishonest majority with preprocessing setting, the amount of preprocessing per party, in the global pool of parties that may be part of committees in the online phase, must be proportional to the size of the global pool. This chapter is based (often verbatim) on two papers. The sections containing the dishonest and honest majority protocols, as well as the dishonest majority lower bound, are based on [Bienstock et al. 2023a]. The section containing the two-thirds honest majority protocol is based on [Bienstock et al. 2023b].

3.1 Security Model and Preliminaries

We present some of the preliminaries required in this chapter. First we discuss the fluid model in Section 3.1.1, and then in Section 3.1.2 we present our security model. We utilize the universal composability framework of [Canetti 2001].

3.1.1 Modelling Fluid MPC

We first recall the modelling of Fluid MPC from [Rachuri and Scholl 2022; Choudhuri et al. 2021]. We consider the *client-server* model, where there is a universe $\mathcal{U}$ of parties, that includes both the clients, who provide inputs, and servers, who perform computation. The goal of these clients is to *privately* compute a function over their inputs. The clients delegate this computation to a set of servers in $\mathcal{U}$ that can volunteer their computational resources for *part* of the computation and then potentially go offline. That is, the set of servers is not fixed in advace, and can change from time to time.

Computation is composed of an optional preprocessing stage among all clients and servers, an input phase where clients *privately* provide inputs, an execution stage where the servers compute the function, and an output phase where the clients receive output. Preprocessing is typically only required to have a statistically-secure execution phase for the dishonest majority setting (see below). In the *preprocessing stage*, all clients and servers in $\mathcal{U}$ interact to generate information that will be used in the execution stage, but that is independent of the actual inputs and the function to be computed (it may be required that *enough* information is generated for some particular function). After the preprocessing stage, servers can go offline until the clients wish to perform the computation. In the *execution stage*, only the servers participate to compute the function. The execution step is itself divided into *epochs*, where each epoch i runs among a fixed set of servers, or committee C_i. An epoch contains two parts, the computation phase, where the committee performs some computation local to itself, followed by a hand-off phase, where the current committee securely transfers some current state to the next committee. We stress that there is only *one output stage*, i.e., the clients get some final state from the servers once that allows them to reconstruct the entire output all at that time. We assume that all parties have access to only point-to-point channels. We refer to the set of clients as C_{clnt} and the last committee that participates in the protocol as C_ℓ. For simplicity, we may assume in some places that $n = n_i = |C_i|$ (i.e., each committee is of the same size).

Fluidity. Both the computation phase and hand-off phase of each epoch in the execution stage may require multiple rounds of interaction. *Fluidity* is defined as the minimum number of rounds in any given epoch of the execution stage. We say a protocol achieves *maximal fluidity* if each epoch i only lasts for one total round. I.e., the computation phase only consists of local computation by the parties in committee C_i, and the hand-off phase consists of only some local computation by the parties in C_i, plus communication from C_i to C_{i+1}. In this paper, we only consider maximal fluidity, as it is the optimal setting to consider and it is the setting considered in the previous works [Rachuri and Scholl 2022; Choudhuri et al. 2021]. However, we stress that in our modelling for maximal fluidity (as well as that of [Rachuri and Scholl 2022; Choudhuri et al. 2021]) the clients in the input and output stage *may interact for a constant number of rounds* (i.e., independent of the circuit depth) to reconstruct the output.

Committee formation. The committees used in each epoch may either be fixed ahead of time, or chosen on-the-fly throughout the execution stage. While fixing committees ahead of time may result in a simpler, more efficient protocol, we focus on the less restrictive, more realistic setting where committees are chosen on-the-fly. This model is more suitable for the goal of making MPC protocols adequate for use over unsable networks since, intuitively, a given committee has better chances of guaranteeing a stable connection if they do not need to commit to a specific online time far in advance. See [Choudhuri et al. 2021] for more motivation and details on committee selection. The model of [Choudhuri et al. 2021] specifies the formation process via an ideal functionality that samples and broadcasts committees according to the desired mechanism. However, as in [Rachuri and Scholl 2022], we desire to divorce the study of committee selection from the actual MPC and simply require that all parties of the current committee C_i somehow agree on the next committee C_{i+1}. Specifically, the parties of committee C_i during the hand-off phase of epoch i (and not before) are informed by the environment $\mathcal{Z}$ of its choice of committee C_{i+1} (i.e., it is a worst-case choice by $\mathcal{Z}$). We make no assumptions or restrictions on the size of committees nor the overlap between committees. In particular, committees may consist of a large number (possibly constant fraction) of parties in the entire universe, $\mathcal{U}$.

Corruptions. We study three different settings for the number of parties that may be corrupted for our model to still require security:

- For *dishonest majority*, the adversary $\mathcal{A}$ may corrupt all-but-one client and all-but-one server in the committee of each epoch. This is the setting that [Rachuri and Scholl 2022] studies.
- For *honest majority*, the adversary $\mathcal{A}$ may only corrupt any minority of servers in the committee of each epoch.[1] This is the setting that [Choudhuri et al. 2021] studies.
- For *two-thirds honest majority*, the adversary $\mathcal{A}$ may only corrupt less than a third of the servers in the committee of each epoch, that is $3t + 1 = n$.

We will sometimes refer to the honest parties of committee C_i as $\mathcal{H}_{C_i}$, and the corrupt parties of committee C_i as $\mathcal{T}_{C_i}$.

We consider a *malicious R-adaptive adversary* from [Choudhuri et al. 2021] and used in [Rachuri and Scholl 2022]. In short, if there is a preprocessing stage, the adversary *statically* chooses some parties to corrupt beforehand. Then, the adversary *statically* chooses a set of clients to corrupt. During each epoch i of the execution phase, after learning which servers are in committee C_i, the

[1] All-but-one client could be corrupted, however.

adversary *adaptively* chooses a subset of C_i to corrupt. Upon such a corruption, the adversary learns the server's entire past state and can send messages on its behalf in epoch i. Therefore, when counting the number of corruptions for some epoch i, we must retroactively include those servers in committee C_i that are corrupted in some later epoch $j > i$. Furthermore, if there is a preprocessing stage, we count a server in committee C_i as corrupted also if they were corrupted during the preprocessing phase.

3.1.2 Security Model

To model Fluid MPC, we adapt the dynamic arithmetic black box (DABB) ideal functionality $\mathcal{F}_{\text{DABB}}$ of [Rachuri and Scholl 2022]. First, we note that our protocols, as written, achieve *security with selective abort* (same as [Rachuri and Scholl 2022; Choudhuri et al. 2021]), where the adversary can prevent any clients of his or her choice from receiving output. However, similar to the protocol of [Choudhuri et al. 2021] (c.f. Appendix A), our protocols can easily achieve *unanimous abort* (in which honest clients either all receive the output or all abort) if the clients have access to a broadcast channel in the last round or if they implement a broadcast over their point-to-point channels. The same applies to the protocol of [Rachuri and Scholl 2022]. Functionality $\mathcal{F}_{\text{DABB}}$, presented below, is parameterized by a finite field $\mathbb{F}_p$, and supports addition and multiplication operations over the field. It keeps track of the current epoch number in a variable i and the committee of the current epoch i in a variable C_i. The functionality receives the identity of the first committee from the clients via input **Init**. During the execution stage, where the current committee may change, the functionality receives the identity of the next committee from the currently active parties via input **Next-Committee** (if it receives inconsistent committees for either of these two inputs, we assume it aborts).

Figure 3.1: Functionality $\mathcal{F}_{\mathsf{DABB}}$

Parameters: Finite field $\mathbb{F}_p$, universe $\mathcal{U}$ of parties, and set of clients $C_{\mathsf{clnt}} \subseteq \mathcal{U}$. The functionality assumes that all parties have agreed upon public identifiers id_x, for each variable x used in the computation.

Init: On input (Init, C) from every party $P_j \in C_{\mathsf{clnt}}$, where each P_j sends the same set $C \subseteq \mathcal{U}$, initialize $i = 1$, $C_1 = C$ as the first active committee. Send (Init, C_1) to the adversary.

Input: On input $(\mathsf{Input}, \mathsf{id}_x, x)$ from some $P_j \in C_{\mathsf{clnt}}$, and $(\mathsf{Input}, \mathsf{id}_x)$ from all other parties in C_{clnt}, store the pair (id_x, x). Send $(\mathsf{Input}, \mathsf{id}_x)$ to the adversary.

Next-Committee: On input $(\mathsf{Next\text{-}Committee}, C)$ from every party $P_j \in C_i$, where each P_j sends the same set $C \subseteq \mathcal{U}$, update $i = i + 1$, $C_i = C$. Send $(\mathsf{Next\text{-}Committee}, C_i)$ to the adversary.

Add: On input $(\mathsf{Add}, \mathsf{id}_z, \mathsf{id}_x, \mathsf{id}_y)$ from every party $P_j \in C_i$, compute $z = x + y$ and store (id_z, z). Send $(\mathsf{Add}, \mathsf{id}_z, \mathsf{id}_x, \mathsf{id}_y)$ to the adversary.

Multiply: On input $(\mathsf{Mult}, \mathsf{id}_z, \mathsf{id}_x, \mathsf{id}_y)$ from every party $P_j \in C_i$, compute $z = x \cdot y$ and store (id_z, z). Send $(\mathsf{Mult}, \mathsf{id}_z, \mathsf{id}_x, \mathsf{id}_y)$ to the adversary.

Output: On input $(\mathsf{Output}, \{\mathsf{id}_{z_m}\})$ from every party $P_j \in C_{\mathsf{clnt}} \cup C_i$, where a value z_m for each id_{z_m} has been stored previously, retrieve $\{(\mathsf{id}_{z_m}, z_m)\}$ and send $(\mathsf{Output}, \{(\mathsf{id}_{z_m}, z_m)\})$ to the adversary. Wait for input from the adversary, and if it is Deliver, send the output to every $P_i \in C_{\mathsf{clnt}}$. Otherwise, abort.

For simplicity, we omit committee selection and tracking from the description of the ideal functionalities presented in the rest of this chapter. However, these components are (implicitly) exactly as presented in $\mathcal{F}_{\mathsf{DABB}}$.

3.1.3 Preliminaries

Notation. We first note that we will often use $l \in C_i$ as shorthand to refer to some party $P_l \in C_i$, and same for some party $P_l \in \mathcal{H}_{C_{i+1}}$ or $P_l \in \mathcal{T}_{C_{i+1}}$.

Layered circuits. We refer the reader to [Choudhuri et al. 2021] for a more precise description on layered circuits. In short, these are arithmetic circuits composed of addition, multiplication and identity gates. The circuit is divided in *layers*, and for each such layer, the inputs to each gate on the layer come directly from the layer above. Every circuit can be made layered by adding enough identity gates.

Secret Sharing. For our dishonest majority protocol, we use additive n-out-of-n secret sharings. We use the notation $[x]^{C}$ to denote such a sharing of a value x between the parties of some committee C.

For our honest and two-thirds honest majority protocols, we will also utilize Shamir secret sharing. We let $[x]_d^{C_i}$ denote a Shamir secret sharing of value x with degree d among the parties of C_i. We let x^j be the j-th share of a Shamir secret sharing $[x]_d^{C_i}$ (typically held by party P_j of committee C_i). As a special case, our protocol will sometimes have a party create a Shamir secret sharing of a share they hold. In this case, we denote that Shamir secret sharing of x^j with degree d among the parties of C_i as $\left[x^j\right]_d^{C_i}$.

We also use the *packed secret sharing* technique introduced by Franklin and Yung [Franklin and Yung 1992] for our two-thirds honest majority protocol. This is similar to Shamir secret sharing, except a vector of ℓ different values $\mathbf{x} = (x_1, \ldots, x_\ell)$ are shared at once using a polynomial that evaluates to $x_1, \ldots, x_\ell$ at ℓ distinct points (w.l.o.g., $-1, \ldots, -\ell$)[2]. For privacy, if t players are corrupted, the polynomial must be random of degree at most $d = t + \ell - 1$. Throughout this chapter, we will use $\ell = t + 1$, and thus $d = 2t$. In this case, similar to standard Shamir secret sharing of degree $d = 2t$, from a set of n shares where at most $t < n/3$ shares are corrupt, their is an algorithm that efficiently either determines the correct vector of secrets, or outputs $\perp$ (denoting an error). We denote a packed secret sharing value of vector $\mathbf{x} = (x_1, \ldots, x_\ell)$ with degree d among the parties of C_i as $[\mathbf{x}]_d^{C_i}$. We let $\mathbf{x}^j$ be the j-th share of a packed secret sharing $[\mathbf{x}]_d^{C_i}$ (typically held

[2]The underlying field size therefore must be at least $\ell + n$.

by party P_j of committee C_i). As a special case, our protocol will sometimes have a party create a Shamir secret sharing of a share of a packed sharing they hold. In this case, we denote that Shamir secret sharing of $\mathbf{x}^j$ with degree d among the parties of C_i as $\left[\mathbf{x}^j\right]_d^{C_i}$. Also, our protocol will sometimes have a party create a packed secret sharing of a vector of shares, each from different Shamir secret sharings, that they hold. In this case, we denote the vector of shares corresponding to sharings $([x_1]_d^{C_i}, \ldots, [x_\ell]_d^{C_i})$ that Party P_j in Committee C_i holds as $\mathbf{x}^j_{[1,\ell]} = (x_1^j, \ldots, x_\ell^j)$ and their packed secret sharing of the vector with degree d among the parties of C_i as $\left[\mathbf{x}^j_{[1,\ell]}\right]_d^{C_i}$.

For our dishonest and honest majority protocols, we will also use *authenticated* secret sharing schemes. For the dishonest majority protcol, we let

$$\langle x \rangle^{i,j} := ((x^i, \Delta^i, M^{i,j}, K^{i,j}), (x^j, \Delta^j, M^{j,i}, K^{j,i}))$$

represent a pairwise BeDOZa [Bendlin et al. 2011] (additive) sharing of x between parties P_i and P_j (i.e., $x = x^i + x^j$), MAC'd under their respective local MAC keys $K^{i,j}, K^{j,i}$ and (additive) shares of some global MAC key Δ^i, Δ^j:

$$M^{i,j} = K^{j,i} + \Delta^j \cdot x^i, \qquad M^{j,i} = K^{i,j} + \Delta^i \cdot x^j.$$

We now let $\langle x \rangle^{\mathcal{P},\mathcal{Q}} := \left(\left(x^i, \{M^{i,j}\}_{j \in \mathcal{Q}}\right)_{i \in \mathcal{P}}, \left(\Delta^j, \{K^{j,i}\}_{j \in \mathcal{Q}}\right)_{i \in \mathcal{P}} \right)$ represent a pairwise BeDOZa [Bendlin et al. 2011] sharing of x where each party P_i of set $\mathcal{P}$ holds an additive share x^i of the secret x and MACs $M^{i,j}$ of this share under the local keys and shares of the global key $K^{j,i}, \Delta^j$ of each party P_j of set $\mathcal{Q}$. Each P_j of $\mathcal{Q}$ indeed holds their share Δ^j of the global MAC key and each local key $K^{j,i}$. These MACs are computed as above.

For both dishonest and honest majority protocols we use another form of authenticated secret sharing, known as SPDZ sharings [Damgård et al. 2012]. For the dishonest majority protocol, an authenticated SPDZ sharing of value x amongst Committee C has the form $[\![x]\!]_{\Delta_C}^C := ([x]^C, \left[\Delta_C \cdot x\right]^C, \left[\Delta_C\right]^C)$, where each sharing component $[\cdot]^C$ is an additive sharing. In this setting, as we will see, each committee C has its own shared MAC key $\Delta_C = \sum_{j \in C_i} \Delta^j$; thus in the above notation, we specify under which committee's MAC key the sharing is authenticated in the subscript. For honest majority, a SPDZ sharing of a value x among the parties of committee C, $\llbracket x \rrbracket^C$ contains a vector of degree-$2t$ Shamir shares $\llbracket x \rrbracket^C := ([x]_{2t}, [\Delta]_{2t}, [\Delta \cdot x]_{2t})$.

Parity Check Matrices. Given a vector $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{F}^n$, we say that $\boldsymbol{x}$ is d-consistent if there exists a polynomial f of degree $\leq d$ such that $f(i) = x_i$ for $i \in [n]$. That is, $\mathbf{x}$ constitutes valid sharings of some secret $x = f(0)$. Testing if a vector $\mathbf{x}$ is d-consistent can be done by interpolating a polynomial of degree $\leq d$ from the first $d+1$ entries, and checking that the remaining $n-(d+1)$ entries are consistent with this polynomial. This can be expressed as a matrix product $(x_{d+2}, \ldots, x_n)^\intercal = \mathbf{H}' \cdot (x_1, \ldots, x_{d+1})$, which can itself be written as $\mathbf{0} = \mathbf{H} \cdot \boldsymbol{x}$, where $\mathbf{H}$ is a $(n-d-1) \times n$ matrix. More formally, $\mathbf{H}$ is the *parity check matrix* of the Reed-Solomon code of length n and dimension $d+1$, and it satisfies that a vector $\mathbf{x} \in \mathbb{F}^n$ is d-consistent if and only if $\mathbf{0} = \mathbf{H} \cdot \mathbf{x} \in \mathbb{F}^{n-(d+1)}$. We will make use of this matrix throughout this chapter.

Super-Invertible and Hyper-Invertible Matrices. In this chapter we make use of super-invertible matrices, and also hyper-invertible matrices [Beerliová-Trubíniová and Hirt 2008], which satisfy that every of its square submatrices is invertible.

3.2 Dishonest Majority Protocol

3.2.1 Technical Overview

We begin by presenting the general idea of our fluid dishonest majority protocol with linear communication complexity, in the preprocessing model. It turns out some of the ideas present in this section will also be helpful for the construction of our fluid honest majority protocol. We then provide a shorter overview for the honest majority protocol in Section 3.4.1.

3.2.1.1 Our Starting Point: Le Mans [Rachuri and Scholl 2022]

We first present the high level ideas of the protocol from [Rachuri and Scholl 2022], which achieves fluid MPC in the dishonest majority setting with quadratic communication complexity. The overall idea is the following. The circuit at hand is considered to be a *layered circuit*. As in [Choudhuri et al. 2021], the invariant that will be kept is that the parties in committee C_i will hold certain sharings of all the current intermediate values in layer i. Eventually, the last committee obtains shares of the outputs of the circuit, which are then transmitted to the clients. Unlike [Choudhuri et al. 2021] however, Le Mans makes use of additive secret-sharing in contrast to Shamir's, due to the setting being dishonest majority in contrast to honest majority.

Resharing and Openings. To maintain the aforementioned invariant, Le Mans makes use of two major blocks. First, to preserve the invariant for addition and identity gates, the parties make use of a *resharing* procedure which enables the parties in committee C_i to transfer additive sharings of some given values $[x_1]^{C_i}, \dots, [x_m]^{C_i}$ to committee C_{i+1} so that, as long as there is at least one honest party in each of these two committees, the adversary learns no information about the underlying secrets, and the parties in C_{i+1} obtain fresh-looking shares $[x_1]^{C_{i+1}}, \dots, [x_m]^{C_{i+1}}$. As we will see later, this resharing can be achieved quite efficiently (in particular, with linear communication complexity) by preprocessing most of the shares that the receiving committee C_{i+1} should hold.

The second major block used in Le Mans is that of Beaver triples, which is preprocessed material of the form $([a]^{C_i}, [b]^{C_i}, [c]^{C_i})$ where $c = a \cdot b$, held by a committee C_i.[3] This enables sharings $[x]^{C_i}, [y]^{C_i}$ held by C_i to be "multiplied", so that committee C_{i+1} obtains sharings $[x \cdot y]^{C_{i+1}}$. This is done by the parties in C_i locally computing $[x + a]^{C_i}$ and $[y + b]^{C_i}$, followed by *opening* these

[3]Recall that a requirement in the fluid preprocessing model is that the correlations the parties receive have to be agnostic to the specific committee assignments. It may not be clear now, but it turns out multiplication triples are committee-agnostic, if the parties start with BeDOZa-style correlations [Bendlin et al. 2011]. This will be made clearer.

sharings towards C_{i+1} by each party in C_i revealing their shares to each party in C_{i+1}. Notice that this takes quadratic communication, and in fact, opening shared values are in essence the exact quadratic bottleneck in Le Mans.

Let us assume temporarily that committee C_{i+1} has sharings of the same multiplication triple, that is, $([a]^{C_{i+1}}, [b]^{C_{i+1}}, [c]^{C_{i+1}})$. After C_{i+1} receives $x + a$ and $y + b$, they can locally compute $[x \cdot y]^{C_{i+1}} \leftarrow (x + a)(y + b) - (y + b)\,[a]^{C_{i+1}} - (x + a)\,[b]^{C_{i+1}} + [c]^{C_{i+1}}$, as required. Now, one way in which committee C_{i+1} could have obtained the multiplication triple is by assuming they obtain it from the preprocessing. However, notice that this triple has to coincide with the one held by C_i, which is harder to achieve while maintaining the requirement of committee-agnostic preprocessing. Instead, in Le Mans the following approach is taken: the parties in C_i *reshare* their triple $([a]^{C_i}, [b]^{C_i}, [c]^{C_i})$ to committee C_{i+1}, which enables the latter committee to obtain $([a]^{C_{i+1}}, [b]^{C_{i+1}}, [c]^{C_{i+1}})$. In principle, using the resharing method sketched above, this can be done with linear communication complexity. However, as we will discuss below, active security demands that besides additive sharings the parties also hold sharings of certain MACs. In Le Mans this is handled by using a different resharing method named key-switching, which makes use of openings and hence it suffers from quadratic communication.

Authenticated Sharings. To prevent a corrupt party from breaking security, Le Mans, as all of the dishonest majority MPC protocols, relies on the SPDZ paradigm [Damgård et al. 2012] of adding authentication to every shared value. This consists of additive sharings of a global MAC key $[\Delta]$, and for each shared value $[x]$, additive sharings of the MAC of this value, computed as $[\Delta \cdot x]$. In the fluid setting, each committee C_i who has shares of a value $[x]^{C_i}$ must also have shares of its MAC $\left[\Delta_{C_i} \cdot x\right]^{C_i}$, together with shares of the global key $\left[\Delta_{C_i}\right]^{C_i}$.

The shared MAC key $\left[\Delta_{C_i}\right]^{C_i}$ can be preprocessed, but it may be different from committee to committee as a result of the committee-agnostic preprocessing condition. Due to this, if the first committee has sharings $([x]^{C_i}, \left[\Delta_{C_i} \cdot x\right]^{C_i}, \left[\Delta_{C_i}\right]^{C_i})$, and the second committee C_{i+1} wants to obtain authenticated sharings $([x]^{C_{i+1}}, [\Delta_{C_{i+1}} \cdot x]^{C_{i+1}}, [\Delta_{C_{i+1}}]^{C_{i+1}})$ under the different key $\Delta_{C_{i+1}}$, this cannot be achieved with the simple resharing from before, given that the secret $\Delta_{C_i} \cdot x$ changes to $\Delta_{C_{i+1}} \cdot x$. This is addressed in Le Mans by using a key switching method (Protocol $\Pi_{\text{Key-Switch}}$ in [Rachuri and Scholl 2022]) that enables an authenticated value under one committee's key to be transferred to the next committee so that it remains authenticated, but under the key of the next committee.

In a bit more detail, assume preprocessed sharings $([r]^{C_i}, [\Delta_{C_{i+1}} \cdot r]^{C_{i+1}})$.[4] With this, given $[\Delta_{C_i} \cdot x]^{C_i}$, committee C_{i+1} can obtain $[\Delta_{C_{i+1}} \cdot x]^{C_{i+1}}$ by letting C_i first compute locally $[x-r]^{C_i}$ and then *opening* this value towards committee C_{i+1}. Then, committee C_{i+1} computes locally $[\Delta_{C_{i+1}} \cdot x]^{C_{i+1}} \leftarrow (x-r) \cdot [\Delta_{C_{i+1}}]^{C_{i+1}} + [\Delta_{C_{i+1}} \cdot r]^{C_{i+1}}$. Again, because this requires opening shared values from one committee to another, the resulting communication complexity is quadratic.

3.2.1.2 The "King Idea" in the Fluid Setting

The reconstruction of a value d requires n^2 communication if all parties just send shares to each other, but it can be done with communication $O(n)$ based on the "king idea" from [Damgård and Nielsen 2007]. This is achieved as follows: in a first round, all share owners send their shares to a single party, a "king", who reconstructs d and sends this value to the intended receiving parties in a second round.

Given that, as we have highlighted above, opening shared values is the bottleneck in Le Mans, a natural approach to achieving linear communication complexity in that protocol is to replace all-to-all openings, which have quadratic communication complexity, by the king idea above. However, this imposes a major complication: all-to-all openings require quadratic communication, but only make use of *one single round*, while in contrast, the king idea has linear communication complexity but requires *two rounds*. As a result, using the king idea does not allow committee C_i

[4]This form of preprocessing is not committee-agnostic, but a simpler form of it is, and the actual tuple required is obtained by adding an extra resharing step. This is not relevant for our discussion.

to open shared values to committee C_{i+1}, but rather, these can be opened towards a committee C_{i+2} (by making use of a king in C_{i+1}). At first sight, one may think that the techniques from Le Mans carry over when using this king idea by simply using two committees per circuit layer, instead of one, to accommodate for the extra round required for the reconstruction of shared values. Unfortunately, as we will argue below, such approach is much more complicated than how it looks at a high level.

Problems with key switching. Recall the key switching protocol from Le Mans sketched above. In that protocol, the parties start with a pair $([r]^{C_i}, \left[\Delta_{C_{i+1}} \cdot r\right]^{C_{i+1}})$, and this enables committee C_i to "transfer" shares of MACs $\left[\Delta_{C_i} \cdot x\right]^{C_i}$ to committee C_{i+1} so that the latter obtains $\left[\Delta_{C_{i+1}} \cdot x\right]^{C_{i+1}}$. This approach works for the one-round openings used in the key switching, but if instead we want to use two-round openings with a king, the king would have to be a member of C_{i+1} itself, and the key switching would have to be done towards committee C_{i+2} instead. This raises a number of complications. First, such approach would require an initial pair $([r]^{C_i}, \left[\Delta_{C_{i+2}} \cdot r\right]^{C_{i+2}})$, but unfortunately such pair is not easily obtainable. The reason is that $\left[\Delta_{C_{i+1}} \cdot r\right]^{C_{i+1}}$ for the inefficient key-switch protocol is obtained in part by the parties of committee C_i using preprocessed "local MACs" of their shares of r under some keys that each party in committee C_{i+1} has. This is allowed in the fluid model, since committee C_i *does* learn the parties of committee C_{i+1} at some point (so that they know to whom to send their sharings of intermediate circuit values). However, committee C_i *never* learns the parties of committee C_{i+2}, so we do not have the preprocessing required to obtain $\left[\Delta_{C_{i+2}} \cdot r\right]^{C_{i+2}}$.

Instead, our approach is to let committee C_{i+2} obtain sharings of the MAC of the secret x, but under a MAC key corresponding to the previous committee C_{i+1}, that is, $\left[\Delta_{C_{i+1}} \cdot x\right]^{C_{i+2}}$. The Le Mans key switching protocol is naturally extended to achieve this by using the king of committee C_{i+1} to reconstruct $(x - r)$ to the parties of committee C_{i+2} and also having committee C_{i+1} reshare $\left[\Delta_{C_{i+1}} \cdot r\right]^{C_{i+1}}$ and $\left[\Delta_{C_{i+1}}\right]^{C_{i+1}}$ with committee C_{i+2}. Committee C_{i+2} can then perform the same computation as committee C_{i+1} did before with these sharings to obtain $\left[\Delta_{C_{i+1}} \cdot x\right]^{C_{i+2}}$. However, with this key switching protocol, we need to take some extra care in our protocol to ensure that MACs of intermediate circuit values do not "fall behind".

In particular, we maintain the invariant that the inputs to the gates at the circuit layer which some committee C_i processes must be authenticated under the MAC key of committee C_{i-2}. For example, $([x]^{C_{i+2}}, \left[\Delta_{C_i} \cdot x\right]^{C_{i+2}})$. This is achieved by preprocessing a multiplication triple where the sharings of a and b are authenticated under the MAC keys of both committees C_{i-2} and C_{i-1}. For example, the a sharing is of the form: $([a]^{C_{i-2}}, \left[\Delta_{C_{i-2}} \cdot a\right]^{C_{i-2}}, \left[\Delta_{C_{i-1}} \cdot a\right]^{C_{i-1}})$.[5] Now, committee C_{i-2} can first reshare the triple that is authenticated under their MAC key Δ_{i-2} to committee C_{i-1}, who can then reshare it to committee C_i. Assuming the invariant holds for committee C_i, it can then successfully compute authenticated sharings of $(x + a)$ and $(y + b)$ under MAC key Δ_{i-2} needed to multiply x and y. Also, from the above resharing by committee C_{i-2}, and the MACs on the triple that committee C_{i-1} already holds, committee C_{i-1} obtains the triple authenticated under their MAC key Δ_{i-1}. Committee C_{i-1} can then key switch the triple with committee C_{i+1} using a king in committee C_i. From this key switching, committee C_{i+1} receives the triple authenticated under the MAC key of committee C_i, e.g., for the a part: $([a]^{C_{i+1}}, \left[\Delta_{C_i} \cdot a\right]^{C_{i+1}})$. Committee C_{i+1} can then reshare this triple to committee C_{i+2}, who can then use the above multiplication technique to obtain $([xy]^{C_{i+2}}, \left[\Delta_{C_i} \cdot (xy)\right]^{C_{i+2}})$, also authenticated under the MAC key of committee C_i. Thus the invariant is preserved.

Authenticating multiplication triples. There is a second and perhaps more subtle problem that arises when using an intermediate king for linear reconstruction. Note that the above preprocessed triples that we can obtain are such that the sharing $[c]^{C_i}$ is *not* authenticated. This is addressed in Le Mans by letting committee C_i learn the authentication of $[c]^{C_i}$, and in fact the whole multiplication triple, from the previous committee C_{i-1}. In a bit more detail, C_{i-1} obtains

[5]This kind of triple authenticated under the MAC keys of both committees C_{i-2} and C_{i-1} can indeed still be computed from our actual committee-agnostic preprocessing.

$([a]^{C_{i-1}}, \left[\Delta_{C_{i-1}} \cdot a\right]^{C_{i-1}}, [b]^{C_{i-1}}, \left[\Delta_{C_{i-1}} \cdot b\right]^{C_{i-1}}, [c]^{C_{i-1}})$ from the preprocessing, and they perform key switching so that committee C_i obtains the multiplication triple with the MAC shares of the factors, only missing the shares of the MAC of c. To obtain $\left[\Delta_{C_i} \cdot c\right]^{C_i}$, a pair $([v]^{C_{i-1}}, \left[\Delta_{C_i} \cdot v\right]^{C_i})$ is generated using the key switch protocol on a preprocessed pair $([v]^{C_{i-1}}, \left[\Delta_{C_{i-1}} \cdot v\right]^{C_{i-1}})$.[6] With the former pair at hand, the parties in C_{i-1} can open $[c - v]^{C_{i-1}}$ to C_i, who can then compute locally $\left[\Delta_{C_i} \cdot c\right]^{C_i} \leftarrow (c - v) \cdot \left[\Delta_{C_i}\right]^{C_i} + \left[\Delta_{C_i} \cdot v\right]^{C_i}$.

We can easily enough tweak our multiplication procedure sketched above so that committee C_{i-2} instead uses a king in committee C_{i-1} to open $(c - v)$ to committee C_i. However, recall that using our key-switch procedure, committee C_i can only obtain sharings from committee C_{i-2} that are authenticated under the MAC key of committee C_{i-1}. But, we need c to be authenticated under the MAC key of committee C_i in order to preserve the invariant described above, since these shares of c are used to compute the shares of the output. Thus, we must wait until committee C_{i+1} to authenticate c under the key of committee C_i. However, since $(x + a)$ and $(y + b)$ are opened to the (possibly corrupt) king of committee C_{i+1}, the adversary could then add errors dependent on x and y to c while authenticating it. The adversary could thus mount a selective failure attack using these errors. To solve this we still use the king technique so that committee C_{i-2} can open $(c - v)$ to committee C_i. We then have *only* some king in committee C_i (to preserve linear communication) again forward $(c - v)$ to committee C_{i+1}, who can then authenticate c. To ensure that this king does not cheat as above, we also have the parties of committee C_i hash the received $(c - v)$ values for *all* multiplication gates at this circuit layer, using a universal hash function. Then the parties of committee C_i send these hashes to *each* party of C_{i+1}, who use them to check consistency of their received openings. Since these hashes are short, in fact independent of the number of gates at this layer, communication is still efficient. We use a similar hashing technique as part of the procedure that checks the MACs of shared values.

[6][Rachuri and Scholl 2022] uses 'l' instead of our 'v' here.

3.2.2 Formal Protocol

The protocol Le Mans from [Rachuri and Scholl 2022] is set in the dishonest majority setting, and they show how to achieve maximal fluidity by relying on preprocessed partially-authenticated multiplication triples, and using accumulators to both verify openings and multiplication correctness. Le Mans, just like the protocol from [Choudhuri et al. 2021], achieves a communication complexity that is quadratic in the size of the committees. However, interestingly, the source of quadratic communication is different for [Rachuri and Scholl 2022]. In [Choudhuri et al. 2021], as we discussed in the previous section, quadratic communication appears in the *resharing* step, where each committee reshares their status of the computation towards the next committee. In contrast, resharing is not a problem in Le Mans, which stems from the fact that they make use of additive secret sharing, which admits for a very efficient resharing protocol if one is willing to assume certain form of preprocessing. Instead, the quadratic complexity in Le Mans appears from the approach they take to secure multiplication, which we expand on below.

As we have already mentioned Le Mans makes use of preprocessed multiplication triples to make progress at every multiplication layer. This reduces the problem of secure multiplication to that of reconstructing a shared value, which they do by letting each party in a given committee send out their shares to every other party in the next committee, which leads to quadratic communication. Instead, we achieve linear communication by handling these reconstructions in a different way: sharings are reconstructed to a single "king", who sends the reconstructions to the parties in the next committee. This is indeed the standard way in which openings are handled in non-fluid dishonest majority such as SPDZ [Damgård et al. 2012], and its derivatives. The details of our protocol are presented below.

Throughout this section we will always use additive n-out-of-n secret sharings. Recall that we use the notation $[x]^{\mathcal{C}}$ to denote such a sharing of a value x between the parties of some committee $\mathcal{C}$.

Remark 3.1 (On the relevance of global preprocessing)**.** *We recall that our fluid modelling allows for the committees to be chosen on the fly, which means that any form of preprocessing has to be agnostic to concrete committee assignments. This is one of the major complications we deal with in this section. If this was not the case, that is, if the preprocessing was allowed to depend on the concrete committee choices, then a much simpler approach can be envisioned: the same authenticated triple is preprocessed across two alternate committees, the first uses it to mask the two secrets to be multiplied, a king in an intermediate committee is used for linear reconstruction, and the other committee uses the same triple to compute the final sharings of the product. This is not a possibility in our case.*

Functionality for commitments. In this section we will make use of the following functionality, also appearing in [Rachuri and Scholl 2022].

Figure 3.2: Functionality $\mathcal{F}_{\text{commit}}$

The functionality runs between a set of parties $\mathcal{P}$ and an adversary $\mathcal{A}$.

Commit: On input $(\text{commit}, P_i, x, \tau_x)$ from P_i, where τ_x is a previously unused identifier, store (P_i, x, τ_x) and sent (P_i, τ_x) to all parties.

Open: On input $(\text{open}, P_i, \tau_x)$ from P_i, retrieve x and send (x, i, τ_x) to all parties.

3.2.2.1 Dishonest Majority Preprocessing

We begin by describing the preprocessing functionality $\mathcal{F}_{\text{prep}}$ that is used for our dishonest majority construction. Functionality $\mathcal{F}_{\text{prep}}$ is in charge of distributing first (additive) shares Δ^j of the global MAC key Δ to each party in the universe $\mathcal{U}$. It also distributes (i) pairwise BeDOZa sharings $\langle r\rangle^{i,j}$ between each pair of parties in $\mathcal{U}$, (ii) partially-authenticated multiplication triple sharings $(\langle a\rangle^{i,j}, \langle b\rangle^{i,j}, [c]^{i,j})$ between each pair of parties in $\mathcal{U}$, and (iii) common random values $s_{i,j}$ shared between each pair of parties in $\mathcal{U}$. For (ii), the shared value c is computed as $a^i \cdot b^j + a^j \cdot b^i$, i.e., the cross terms in the product $(a^i + a^j) \cdot (b^i + b^j) = a \cdot b$ of the values a and b for which the two parties have pairwise BeDOZa sharings. Note that $[c]^{i,j}$ is not authenticated and in fact can have additive error of the form $\delta_a \cdot b^j + \delta_b \cdot a^j$, if P_i is corrupted and P_j is honest.

We remark that this is a functionality that we do not aim at instantiating in our work. We refer the reader to [Rachuri and Scholl 2022] on how this functionality can be instantiated by having the pool of all parties perform an MPC protocol among themselves. We note that this instantiation requires per-party communication and storage of $O(N \cdot \log|C|)$, if each party may participate in a constant fraction of committees in the worst-case during the execution phase. However, in order to have a purely *information-theoretic* execution phase, the stored preprocessing states from [Rachuri and Scholl 2022] must be expanded to size $\Theta(N \cdot |C|)$, *before* the execution phase, consistent with our lower bound from Section 3.3.

Figure 3.3: Functionality $\mathcal{F}_{\mathsf{prep}}$

Parameters: Finite fields $\mathbb{F}_p$ and $\mathbb{F}_{p^r}$, parties $P_1, \ldots, P_N$, adversary $\mathcal{A}$, set of honest parties Hon, and set of corrupt parties Corr.

Functionality: Generate pairwise authenticated random values and pairwise partially-authenticated multiplication triples.

1. $\mathcal{F}_{\mathsf{prep}}$ receives from $\mathcal{A}$ the global MAC key shares $\{\Delta^i\}_{i \in \mathsf{Corr}}$ for the corrupt parties. It then samples randomly $\Delta^j \leftarrow_\$ \mathbb{F}_{p^r}$ for each honest party $P_j \in \mathsf{Hon}$.

2. For $2 \cdot \texttt{tot\_trip} + \texttt{tot\_rand}$ number of times, for every $P_i \neq P_j$:

    (a) If both $P_i, P_j \in \mathsf{Hon}$, $\mathcal{F}_{\mathsf{prep}}$ samples random values $r^i, r^j \leftarrow_\$ \mathbb{F}_p$ and MAC keys $K^{i,j}, K^{j,i} \leftarrow_\$ \mathbb{F}_{p^r}$. It then computes $M^{i,j} \leftarrow K^{j,i} + \Delta^j \cdot r^i \in \mathbb{F}_{p^r}$ and $M^{j,i} \leftarrow K^{i,j} + \Delta^i \cdot r^j \in \mathbb{F}_{p^r}$.

    (b) If one of P_i or P_j, say P_i w.l.o.g., is in Corr, $\mathcal{F}_{\mathsf{prep}}$ receives from $\mathcal{A}$ share r^i along with MAC $M^{i,j}$ and MAC key $K^{i,j}$. It also samples random share $r^j \leftarrow_\$ \mathbb{F}_p$, then computes $M^{j,i} \leftarrow K^{i,j} + \Delta^i \cdot r^j$ and $K^{j,i} \leftarrow M^{i,j} - \Delta^j \cdot r^i$.

    (c) If both P_i and P_j are in Corr, $\mathcal{F}_{\mathsf{prep}}$ receives from $\mathcal{A}$ all corresponding values.

3. For `tot_trip` number of times, for every $P_i \neq P_j$, $\mathcal{F}_{\text{prep}}$ let $\langle a_{m_T} \rangle^{i,j}, \langle b_{m_T+1} \rangle^{i,j}$ be the outputs from the m_T and (m_T+1)-st iterations of Step 2 above. Then:

   (a) If both $P_i, P_j \in \text{Hon}$, $\mathcal{F}_{\text{prep}}$ samples $c^{i,j}, c^{j,i} \in \mathbb{F}_p$ randomly such that $c^{i,j} + c^{j,i} = a^i_{m_T} \cdot b^j_{m_T+1} + a^j_{m_T} \cdot b^i_{m_T+1}$.

   (b) If one of P_i or P_j, say P_i w.l.o.g., is in Corr, $\mathcal{F}_{\text{prep}}$ receives from $\mathcal{A}$ share $c^{i,j}$ and additive errors δ_a, δ_b. It then computes $c^{j,i} \leftarrow \Big((a^i_{m_T} + \delta_a) \cdot b^j_{m_T+1} + a^j_{m_T} \cdot (b^i_{m_T+1} + \delta_b)\Big) - c^{i,j}$.

   (c) If both P_i and P_j are in Corr, $\mathcal{F}_{\text{prep}}$ receives from $\mathcal{A}$ all corresponding values.

4. For `tot_shared_rand` number of times, for every $P_i \neq P_j$:

   (a) If both $P_i, P_j \in \text{Hon}$, $\mathcal{F}_{\text{prep}}$ samples random $s_{i,j} \leftarrow_\$ \mathbb{F}_p$.

   (b) If at least one of P_i or P_j, say P_i w.l.o.g., is in Corr, $\mathcal{F}_{\text{prep}}$ receives $s_{i,j}$ from $\mathcal{A}$.

5. Finally, $\mathcal{F}_{\text{prep}}$ outputs all shares of $\langle r \rangle^{i,j}$, $(\langle a \rangle^{i,j}, \langle b \rangle^{i,j}, [c]^{i,j})$, and $s_{i,j}$ computed above to parties in Hon.

Functionality $\mathcal{F}_{\text{prep}}$ is useful as it can be used to generate partially authenticated multiplication triples and shares of authenticated uniformly random values within any committee, once the identity of this committee is known. This is described in detail in Procedure $\pi_{\text{get-combined-prep}}$.

Figure 3.4: Procedure $\pi_{\text{get-combined-prep}}$

Usage: Generate BeDOZa random sharings and partially-authenticated BeDOZa random multiplication triples using the pairwise random sharings and partially-authenticated random multiplication triples from $\mathcal{F}_{\text{prep}}$.

Init: m_T is initialized to 0 and m_R is initialized to $2 \cdot$ `tot_trip`.

1. On input $(\text{rand}, \mathcal{P}, \mathcal{Q})$, each party $P_i \in \mathcal{P}$ outputs $(r^i_{m_R}, \{M^{i,j}_{m_R}\}_{j \in \mathcal{Q}})$ and each $P_j \in \mathcal{Q}$ outputs $(\Delta^j, \{K^{j,i}_{m_R}\}_{i \in \mathcal{P}})$, using the fresh m_R-th rand value from $\mathcal{F}_{\text{prep}}$. Then m_R is incremented: $m_R \leftarrow m_R + 1$.

2. On input $(\mathrm{trip}, \mathcal{P}, \mathcal{Q})$, each party $P_i \in \mathcal{P}$ outputs $((a^i_{m_T}, \{M^{i,j}_{m_T}\}_{j\in\mathcal{Q}}), (b^i_{m_T+1}, \{M^{i,j}_{m_T+1}\}_{j\in\mathcal{Q}}), a^i_{m_T} \cdot b^i_{m_T+1} + \sum_{l\in\mathcal{P}\setminus\{i\}} c^{i,l}_{m_T})$ and each $P_j \in \mathcal{Q}$ outputs $(\Delta^j, \{K^{j,i}_{m_T}\}_{i\in\mathcal{P}}, \{K^{j,i}_{m_T+1}\}_{i\in\mathcal{P}})$, using the fresh m_T-th partially authenticated triples from $\mathcal{F}_{\mathrm{prep}}$. Then m_T is incremented: $m_T \leftarrow m_T + 2$.

3.2.2.2 Efficient Resharing for Dishonest Majority

For efficient resharing we make use of the techniques in [Rachuri and Scholl 2022], which consist of the parties sending additive shares of their shares towards the next committee, but using some preprocessing in the form of pairwise shared randomness in order to precompute most of the shares.[7] Details are given in Procedure $\pi_{\text{eff-reshare-dm}}$ below.

Figure 3.5: Procedure $\pi_{\text{eff-reshare-dm}}$

Usage: C_i reshares $[r]^{C_i}$ to C_{i+1}.

1. Each party P_j in C_i will use the next fresh pairwise shared random values from $\mathcal{F}_{\mathrm{prep}}$, $\{r_{j,l}\}_{l\in C_{i+1}}$.
2. Each P_j will then take their share r^j of $[r]^{C_i}$ and locally compute $r^{j,1} = r^j - \sum_{l=2}^{n_{i+1}} r_{j,l}$.
3. Next, each P_j will send their $r^{j,1}$ to P_1 in C_{i+1}.
4. Finally, parties P_l in C_{i+1} will locally compute their share r^l of $[r]^{C_{i+1}}$ as $r^l \leftarrow \sum_{j\in C_i} r^{l,j}$ (where if $j \neq 1$, each $r^{l,j} = r_{l,j}$, obtained from $\mathcal{F}_{\mathrm{prep}}$).

Lemma 3.2. *Procedure $\pi_{\text{eff-reshare-dm}}$'s transcript is simulatable by random values.*

Proof. Assume without loss of generality that only party $P_{n_i} \in C_i$ is honest (the case where other parties are honest follows easily). Now $P_1 \in C_{i+1}$ is the only party that receives communication in

[7]The parties can instead use pairwise shared PRG seeds, and expand them each time they need the values $r_{j,l}$ used in $\pi_{\text{eff-reshare-dm}}$.

this procedure, so if P_1 is not corrupted, it is easy to simulate. If P_1 is corrupted, assume without loss of generality that $P_{n_{i+1}}$ is honest (again, the case where others are honest follows easily). So, we must argue that simulating $r^{n_i,1}$, that $P_1 \in C_{i+1}$ receives from $P_{n_i} \in C_i$, with a random value is valid. From $\mathcal{F}_{\text{prep}}$, we know that only P_{n_i} and $P_{n_{i+1}}$ have knowledge of $r_{n_i,n_{i+1}}$; for everyone else, it is distributed uniformly at random. Thus, since $r_{n_i,n_{i+1}}$ is added to $r^{n_i,1}$, $r^{n_i,1}$ appears uniformly random to $\mathcal{A}$, and we are done. □

As in Le Mans, we also rely on the fact that BeDOZa authenticated sharings can be converted locally into SPDZ sharings. This is carefully discussed in Procedure π_{convert}. First, recall that an authenticated SPDZ sharing of value x amongst Committee C has the form $[\![x]\!]^C_{\Delta_C} := ([x]^C, \left[\Delta_C \cdot x\right]^C, \left[\Delta_C\right]^C)$. Since each committee C has its own shared MAC key $\Delta_C = \sum_{j \in C_i} \Delta^j$, we specify under which committee's MAC key the sharing is authenticated in the subscript.

Figure 3.6: Procedure π_{convert}

Usage Case 1: For input pairwise-authenticated BeDOZa sharing $\langle x \rangle^{C_i, C_i}$, convert to SPDZ sharing $[\![x]\!]^{C_i}_{\Delta_i}$.

1. Note that
$$\sum_{j \in C_i} \left(\Delta^j \cdot x^j + \sum_{l \in C_i} M^j_l - K^j_l \right) =$$
$$\sum_{j \in C_i} \left(\Delta^j \cdot x^j + \sum_{l \in C_i} K^l_j + \Delta^l \cdot x^j - K^j_l \right) = \Delta_{C_i} \cdot x.$$

2. So, each P_j in C_i outputs as their SPDZ share of x: $(x^j, \Delta^j \cdot x^j + \sum_{l \in C_i} M^j_l - K^j_l, \Delta^j)$.

Usage Case 2: For input pairwise-authenticated BeDOZa sharing $\langle x \rangle^{C_i, C_i \cup C_{i+1}}$, convert to SPDZ sharings $[\![x]\!]^{C_i}_{\Delta_i}$ *and* $[\![x]\!]^{C_{i+1}}_{\Delta_{i+1}}$.

1. We first note that C_i can obtain SPDZ shares of x in the same way as above (by ignoring its pairwise MAC and MAC keys with those parties $P_l \in C_{i+1}$, as the sum above is written).

2. Now, each $P_j \in C_i$ additionally computes $M^j \leftarrow \sum_{l \in C_{i+1}} M^j_l$ to obtain sharing $[M]^{C_i}$ and

invokes $\pi_{\text{eff-reshare-dm}}$ on it, along with $[x]^{C_i}$.

3. Let M^l be the share of $[M]^{C_{i+1}}$ obtained by P_l. Now note that

$$\sum_{l \in C_{i+1}} \left(M^l - \sum_{j \in C_i} K_j^l \right) =$$

$$\sum_{l \in C_{i+1}} \left(M^l - \sum_{j \in C_i} M_l^j - \Delta^l \cdot x^j \right) = 0 + \Delta_{C_{i+1}} \cdot x.$$

4. So, each P_l in C_{i+1} outputs as their SPDZ share of x: $(x^l, M^l - \sum_{j \in C_i} K_j^l, \Delta^l)$.

Lemma 3.3. *Procedure π_{convert}'s transcript is simulatable by random values.*

Proof. Follows from Lemma 3.2, since the only communication is invoking $\pi_{\text{eff-reshare-dm}}$ on $[M]^{C_i}$, for which the honest shares are uniformly random, by the security of $\mathcal{F}_{\text{prep}}$. □

3.2.2.3 Checking and Maintaining MACs

Similar to our honest majority protocol of Section 3.4 (and Le Mans), in our dishonest majority protocol, we use n-out-of-n sharings. This means that a malicious adversary can easily add errors to *any* value throughout the computation. Thus, as before, we use MACs, which authenticate every intermediate value used throughout the computation, and guarantee those values' integrity. To attest to the integrity of every value that is opened throughout the protocol, we again use an accumulator that is computed similarly as before. As in the honest majority case, to ensure that the adversary does not cheat, we open a challenge β to the parties of C_{i+1}, even though the values are opened to C_i. The parties of C_{i+1} then use it to compute a random linear combination on the openings. So, to maintain linear communication, P_1 of C_i forwards the openings to every party in C_{i+1}, and we ensure that P_1 does not cheat by having all of the other parties of C_i send hashes of the openings to the parties of C_{i+1}. Since the hashes are short, total communication will still be $O(n|C|)$ if the width of C is $\Omega(n)$. Details are given in Procedure $\pi_{\text{MAC-check-dm}}$ below.

Figure 3.7: Procedure $\pi_{\text{MAC-check-dm}}$

Usage: Each committee C_i incrementally updates a MAC check state $[\sigma]^{C_i}$ based on the values $\{[\![A_m]\!]^{C_{i-2}}_{\Delta_j}\}_{m\in[T]}$ (for some $j \le i-2$) opened to them, which the final committees at the end of the computation use to check that all openings throughout the protocol were performed correctly.

Init: Each Party P_i in committee C_2 (the first to have values opened to it) initially defines their share of $[\sigma]^{C_2}$ as $\sigma^i \leftarrow 0$.

Update State: On input (update, $\{(A_m, \left[A_m \cdot \Delta_j\right]^{C_i})\}_{m\in[T]}, \left[\Delta_j\right]^{C_i}$) from committee C_i, where $\{A_m\}_{m\in[T]}$ were the values opened to C_i:

1. Committee C_i invokes $\pi_{\text{get-combined-prep}}$ with $(\text{rand}, C_i, C_{i+1})$ so that $P_j \in C_i$ gets $(\beta^j, \{M^{j,l}\}_{l\in C_{i+1}})$ and $P_l \in C_{i+1}$ gets $(\Delta^l, \{K^{l,j}\}_{j\in C_i})$.
2. Each $P_j \in C_i$ also interprets the next pairwise shared random values from $\mathcal{F}_{\text{prep}}$, $\{s_{j,l}\}_{l\in C_{i+1}}$ as keys to a universal hash family $\mathsf{H} = \{\mathsf{H}_s : \mathbb{F}_p^T \to \mathbb{F}_p\}$ and computes $h_{j,l} \leftarrow \mathsf{H}_{s_{j,l}}(\{A_m\}_{m\in[T]})$ for each $P_l \in C_{i+1}$.
3. In parallel: (i) each party $P_j \in C_i$ then sends to each $P_l \in C_{i+1}$ their share β^j and corresponding MAC for P_l, $M^{j,l}$, along with the computed hash value $h_{j,l}$; (ii) only $P_1 \in C_i$ sends $\{A_m\}_{m\in[T]}$ to each $P_l \in C_{i+1}$; and (iii) all of C_i invokes $\pi_{\text{eff-reshare-dm}}$ on $[\sigma]^{C_i}, \left[\Delta_{C_j}\right]^{C_i}, \{\left[\Delta_{C_j} \cdot A_m\right]^{C_i}\}_{m\in[T]}$.
4. Each $P_l \in C_{i+1}$ then locally checks that $M^{j,l} \leftarrow \beta^j \cdot \Delta^l + K^{l,j}$, for each $P_j \in C_i$, and aborts if any check fails. If not, let $\beta \leftarrow \sum_{j\in C_i} \beta^j$.
5. Each P_l additionally uses the pairwise shared random values from $\mathcal{F}_{\text{prep}}$, $\{s_{j,l}\}_{j\in C_i}$, to compute $h'_{j,l} \leftarrow \mathsf{H}_{s_{j,l}}(\{A_m\}_{m\in[T]})$, checks that each $h'_{j,l} = h_{j,l}$, and aborts if any check fails; else continues.
6. Each $P_l \in C_{i+1}$ next locally computes $A \leftarrow \sum_{m=1}^{T}(\beta)^m \cdot A_m$ and $[\gamma]^{C_{i+1}} \leftarrow \sum_{m=1}^{T}(\beta)^m \cdot \left[\Delta_{C_j} \cdot A_m\right]^{C_{i+1}}$ (here $(\beta)^m$ is the m-th power of β).

7. It finally updates $[\sigma]^{C_{i+1}} \leftarrow [\sigma]^{C_{i+1}} + [\gamma]^{C_{i+1}} - \left[\Delta_{C_j}\right]^{C_{i+1}} \cdot A$ and invokes $\pi_{\text{eff-reshare-dm}}$ on $[\sigma]^{C_{i+1}}$.

Check State: On input check from the clients C_{clnt}:

1. Each client $P_i \in C_{\text{clnt}}$ invokes $\mathcal{F}_{\text{commit}}$ on their share σ^i of the MAC check state $[\sigma]^{C_{\text{clnt}}}$.
2. Then they all open their commitments, and if they are consistent, output Accept if $\sum_{i \in C_{\text{clnt}}} \sigma^i = 0$; else Reject.

Lemma 3.4. *Procedure $\pi_{\text{MAC-check-dm}}$ is correct, i.e., it accepts if all the opened values A_m and the corresponding MACs are computed correctly. Moreover, it is sound, i.e., it rejects except with probability at most $(2 + \max_i T_i)/p$ in case at least one opened value is not correctly computed. Furthermore, the transcript of* ***Update State*** *is simulatable.*

Proof. For soundness, we consider all of the points in which the adversary can inject error, when a single committee C_i is updating $[\sigma]^{C_i}$. First, note that with all-but-negligible probability, an honest party P_j of C_i will receive the same (potentially incorrect) $A'_m = A_m + \delta^i_m$ for each m as an honest party P_l of C_{i+1}. This is because their shared universal hash key $s_{j,l}$ from $\mathcal{F}_{\text{prep}}$ is uniformly random and unknown to the adversary. So, since H is a universal hash family, it holds that if P_j gets $\{A'_m\}_{m\in[T]}$ and P_l gets a different $\{A''_m\}_{m\in[T]}$,

$$\Pr[\mathsf{H}_{s_{j,l}}(\{A'_m\}_{m\in[T]}) = h_{j,l} = h'_{j,l} = \mathsf{H}_{s_{j,l}}(\{A''_m\}_{m\in[T]})] \leq 1/p.$$

The adversary can thus only inject the following kind of errors:

1. When opening shares of some A_m to P_{king} of C_{i-1}, or if P_{king} itself is corrupted, then when sending opened A_m to the honest parties P_j of committee C_i, the adversary can add some δ^i_m error. From above, these δ^i_m's must in fact be independent of the opened challenge randomness for C_i, β^i.

2. When the MAC key $\left[\Delta_{C_j}\right]^{C_i}$ is reshared, the adversary can add some additive η^i error.

3. When resharing a MAC $\left[\Delta_{C_j} \cdot A_m\right]^{C_i}$, the adversary can add some additive ε^i_m.

4. Finally, when $[\sigma]^{C_{i+1}}$ is reshared, the adversary can add some additive ζ^i.

So, at the end of the computation (after ℓ committees), if the check passes we will have:

$$0 = [\sigma]^{C_{\mathsf{clnt}}} = \sum_{i=1}^{\ell}\sum_{m=1}^{T_i}(\beta^i)^m \cdot ((\Delta_{C_{j_i}} \cdot A^i_m + \varepsilon^i_m) - (\Delta_{C_{j_i}} + \eta^i)\cdot(A^i_m + \delta^i_m)) + \zeta^i$$

$$= \sum_{i=1}^{\ell}\sum_{m=1}^{T_i}(\beta^i)^m \cdot (\varepsilon^i_m - \Delta_{C_{j_i}} \cdot \delta^i_m + \eta^i \cdot A^i_m + \delta^i_m \cdot \eta^i) + \zeta^i$$

Recall that we want to show that for any i, m, $\delta^i_m \neq 0$ can only happen with negligible probability. First, note that the corrupt parties of committee C_i cannot forge their share β^j of β to any of the honest parties of C_{i+1} except with probability $1/p$, by the security of the information-theoretic MAC provided by $\mathcal{F}_{\mathsf{prep}}$. Thus, the reconstructed challenge β must indeed be uniformly random and independent of all other values. Now, assume that indeed there is some $\delta^{i^*}_m \neq 0$. First, we argue that by the Schwartz-Zippel Lemma, with probability $\leq T_{i^*}/p$, $\sum_{m=1}^{T_{i^*}}(\beta^{i^*})^m \cdot \delta^{i^*}_m = 0$. This holds since β^{i^*} is chosen uniformly at random and, by assumption, at least one $\delta^{i^*}_m \neq 0$. So now we rewrite:

$$0 = \sum_{i=1}^{\ell}\sum_{m=1}^{T_i}(\beta^i)^m \cdot (\varepsilon^i_m - \Delta_{C_{j_i}} \cdot \delta^i_m + \eta^i \cdot A^i_m + \delta^i_m \cdot \eta^i) + \zeta^i$$

$$= -\Delta_{C_{j_{i^*}}} \cdot \sum_{m=1}^{T_{i^*}}(\beta^{i^*})^m \cdot \delta^{i^*}_m + \sum_{m=1}^{T_{i^*}}(\beta^{i^*})^m \cdot (\varepsilon^{i^*}_m + \eta^{i^*} \cdot A^{i^*}_m + \delta^{i^*}_m \cdot \eta^{i^*}) + \zeta^{i^*} +$$

$$\sum_{i\in[\ell]\setminus\{i^*\}}\sum_{m=1}^{T_i}(\beta^i)^m \cdot (\varepsilon^i_m - \Delta_{C_{j_i}} \cdot \delta^i_m + \eta^i \cdot A^i_m + \delta^i_m \cdot \eta^i) + \zeta^i.$$

Therefore:

$$\Delta_{C_{j_{i^*}}} = \frac{1}{\sum_{m=1}^{T_{i^*}} (\beta^{i^*})^m \cdot \delta_m^{i^*}} \left(\sum_{m=1}^{T_{i^*}} (\beta^{i^*})^m \cdot (\varepsilon_m^{i^*} + \eta^{i^*} \cdot A_m^{i^*} + \delta_m^{i^*} \cdot \eta^{i^*}) + \zeta^{i^*} + \right.$$

$$\left. \sum_{i \in [\ell] \setminus \{i^*\}} \sum_{m=1}^{T_i} (\beta^i)^m \cdot (\varepsilon_m^i - \Delta_{C_{j_i}} \cdot \delta_m^i + \eta^i \cdot A_m^i + \delta_m^i \cdot \eta^i) + \zeta^i \right).$$

However, since $\Delta_{C_{j_{i^*}}}$ is sampled uniformly at random and independently, and is unknown to the adversary, this can only happen with probability $1/p$.

In total, the probability that there is some $\delta_m^i \neq 0$ is bounded by $(2 + \max_i T_i)/p$, which is negligible.

Correctness clearly holds if all errors are 0. Furthermore, we know from Lemma 3.2 that $\pi_{\text{eff-reshare-dm}}$ is simulatable by random values. Also, the simulator knows the universal hash keys that honest parties use to compute the hashes on the opened values (which it also knows), thus it can simulate these hashes itself. Furthermore, each β^j is sampled uniformly at random in $\mathcal{F}_{\text{prep}}$, so the simulator can also simulate these, and the corresponding MACs by using the MAC keys obtained from $\mathcal{F}_{\text{prep}}$. Finally, the simulator can easily emulate $\mathcal{F}_{\text{commit}}$ for the commitments. □

In our protocol, each committee C_i has their own MAC key Δ_{C_i}. Thus, when some state $[\![x]\!]_{\Delta_i}^{C_i}$ of the computation is reshared from one committee to the next, we also need to somehow "switch" the key of its MAC to that of the new committee. Le Mans also needs to deal with this issue. To do so, they take advantage of π_{convert} to obtain random sharings $[\![t]\!]_{\Delta_i}^{C_i}, [\![t]\!]_{\Delta_{i+1}}^{C_{i+1}}$ of the same value, authenticated under *both* committees' MAC keys. This is sufficient for them, as they can then mask $[\![x]\!]_{\Delta_i}^{C_i}$ with t, reconstruct $x + t$, and then use their two authenticated sharings of t, $[\![t]\!]_{\Delta_i}^{C_i}, [\![t]\!]_{\Delta_{i+1}}^{C_{i+1}}$ (including some relationship between their MACs), to create unmasked shares of x that are indeed authenticated under Δ_{i+1}. Note, however, that they use reconstruction of $x + t$ from one committee to the next, which has quadratic communication.

We instead use the "king idea" to reconstruct $(x + t)$ across two committees, i.e., from C_i to C_{i+2}.

However, since π_{convert} can only obtain authenticated random sharings of the same random value between *consecutive* committees, $\pi_{\text{eff-key-switch}}$ below switches $[\![x]\!]^{C_i}_{\Delta_i}$, shared and authenticated among C_i, to $[\![x]\!]^{C_{i+2}}_{\Delta_{i+1}}$, shared amongst C_{i+2}, but authenticated using C_{i+1}'s key.

Figure 3.8: Procedure $\pi_{\text{eff-key-switch}}$

Usage: Transform a SPDZ sharing $[\![x]\!]^{C_i}_{\Delta_i}$ held by C_i and MAC'd under C_i's key to a SPDZ sharing $[\![x]\!]^{C_{i+2}}_{\Delta_{i+1}}$ held by C_{i+2} and MAC'd under C_{i+1}'s key.

1. Let $[\![x]\!]^{C_i}_{\Delta_i}$ be the shares of the input authenticated value x. All parties in C_i agree on a special party P_{king} in C_{i+1}.
2. C_i and C_{i+1} invoke $\pi_{\text{get-combined-prep}}$ with $(\text{rand}, C_i, C_i \cup C_{i+1})$ to receive $\langle t \rangle^{C_i, C_i \cup C_{i+1}}$.
3. C_i and C_{i+1} then invoke π_{convert} on $\langle t \rangle^{C_i, C_i \cup C_{i+1}}$ to form $[\![t]\!]^{C_i}_{\Delta_i}$ and $[\![t]\!]^{C_{i+1}}_{\Delta_{i+1}}$.
4. Parties in C_i in parallel: (i) invoke $\pi_{\text{eff-reshare-dm}}$ on input $[x]^{C_i}$; and (ii) open shares of $[\![(x+t)]\!]^{C_i}_{\Delta_i}$ to P_{king} in C_{i+1}.
5. While P_{king} of C_{i+1} distributes opened value $(x+t)$ to all parties in C_{i+2}, all parties in C_{i+1} invoke $\pi_{\text{eff-reshare-dm}}$ on input $[x]^{C_{i+1}}$, $\left[\Delta_{C_{i+1}}\right]^{C_{i+1}}$, and $\left[\Delta_{C_{i+1}} \cdot t\right]^{C_{i+1}}$.
6. Finally, each party P_j in C_{i+2} locally computes its share of the MAC $\left[\Delta_{C_{i+1}} \cdot x\right]^{C_{i+2}}$ as $\left[\Delta_{C_{i+1}}\right]^{C_{i+2}} \cdot (x+t) - \left[\Delta_{C_{i+1}} \cdot t\right]^{C_{i+2}}$. C_{i+2} outputs $\left([x]^{C_{i+2}}, \left[\Delta_{C_{i+1}} \cdot x\right]^{C_{i+2}}, \left[\Delta_{C_{i+1}}\right]^{C_{i+2}}\right)$.

Lemma 3.5. *Procedure $\pi_{\text{eff-key-switch}}$'s transcript is simulatable by random values.*

Proof. First, from Lemmas 3.2 and 3.3, we know that all values communicated to $\mathcal{A}$ by π_{convert} and $\pi_{\text{eff-reshare-dm}}$ are simulatable by random values. Furthermore, from $\mathcal{F}_{\text{prep}}$, we know that $[\![t]\!]^{C_i}_{\Delta_i}$ is a random sharing of a random value. Thus, also the openings $(x+t)$ can be simulated with random values. □

We leverage $\pi_{\text{eff-key-switch}}$ in the following procedure, $\pi_{\text{get-x-comm-shrs}}$, to obtain SPDZ sharings of the same random value amongst *both* committees C_i and C_{i+3}. Committee C_i's sharing will be authenticated under its own key, while committee C_{i+3}'s sharing will be authenticated under C_{i+2}'s key.

Figure 3.9: Procedure $\pi_{\text{get-x-comm-shrs}}$

Usage: On input BeDOZa sharing $\langle r\rangle^{C_i, C_i \cup C_{i+1}}$, output $[\![r]\!]^{C_i}_{\Delta_i}$ to C_i and $[\![r]\!]^{C_{i+3}}_{\Delta_{i+2}}$ to C_{i+3}.

1. C_i and C_{i+1} invoke π_{convert} on $\langle r\rangle^{C_i, C_i \cup C_{i+1}}$ to obtain $[\![r]\!]^{C_i}_{\Delta_i}$ and $[\![r]\!]^{C_{i+1}}_{\Delta_{i+1}}$.
2. C_i outputs $[\![r]\!]^{C_i}_{\Delta_i}$.
3. C_{i+1} then invokes $\pi_{\text{eff-key-switch}}$ on $[\![r]\!]^{C_{i+1}}_{\Delta_{i+1}}$ with C_{i+3} (through C_{i+2}), who outputs $[\![r]\!]^{C_{i+3}}_{\Delta_{i+2}}$.

Lemma 3.6. *Procedure $\pi_{\text{get-x-comm-shrs}}$'s transcript is simulatable by random values.*

Proof. Since only π_{convert} and $\pi_{\text{eff-key-switch}}$ are invoked, we know from Lemmas 3.3 and 3.5 that the transcript is simulatable by random values. □

3.2.2.4 Secure Multiplication and Verification

Finally, before we present the complete protocol, we present Procedure $\pi_{\text{mult-dm}}$ below which enables a given committee to make progress on the computation by securely processing multiplication gates. As in the honest majority protocol of Section 3.4, this procedure makes use of multiplication triples to reduce the task of securely multiplying two shared values to that of reconstructing two secrets. Once again, we use the "king idea" to achieve reconstruction with linear communication.

As in the honest majority case, we have two issues to deal with. First multiplication triples require different committees to have access to the same triple. This is *in part* achieved by making use of the efficient resharing procedure $\pi_{\text{eff-reshare-dm}}$. Also, a given committee can once again only obtain a *partially* authenticated multiplication triple $([\![a]\!]^{C_i}_{\Delta_i}, [\![b]\!]^{C_i}_{\Delta_i}, [c]^{C_i})$ from $\mathcal{F}_{\text{prep}}$, where the c part is not authenticated (and in fact might have some error). Using the same idea from [Rachuri and

Scholl 2022] and Section 3.4, we authenticate the c part "on the fly" using a random, authenticated sharing $[\![v]\!]_{\Delta_i}^{C_i}$.

However, since our $\pi_{\text{eff-key-switch}}$ procedure when resharing a value from C_i to C_{i+2} only switches the MAC key to that of C_{i+1}, we need to take some extra care to ensure that the MAC does not "fall behind". In particular, $\pi_{\text{mult-dm}}$ below maintains the invariant that the inputs to multiplication gates at some circuit level which C_i computes are MAC'd under C_{i-2}'s key. At a high-level, this is accomplished by combining the above "on the fly" authentication of triples obtained from $\mathcal{F}_{\text{prep}}$ by C_{i-2}, so that the triple is MAC'd under the same key as the inputs, with the use of $\pi_{\text{get-x-comm-shrs}}$ to transfer this triple to C_{i+1} (under C_i's key). C_{i+1} can then reshare this triple to C_{i+2}, who can then use the usual multiplication triple technique to compute the output of the multiplication gate, MAC'd under C_i's key, thus maintaining the invariant.

Unfortunately, we are not done yet. Committee C_i can only obtain outputs that are MAC'd under C_{i-1}'s key when C_{i-2} tries to use $\pi_{\text{eff-key-switch}}$ on its sharings. Thus, in order to maintain the invariant above, we need to wait until C_{i+1} to authenticate the c part of multiplication triples. However, this would allow the adversary to add errors dependent on x and y to c, since $(x + a)$ and $(y + b)$ are opened from C_i to the king of C_{i+1}, leading to a selective failure attack. To solve this, the value $(c + v)$ needed to authenticate c is first opened to the parties of C_i using the efficient "king idea", then *only* P_1 of C_i sends the opened $(c + v)$ to the parties of C_{i+1}, for efficiency. To ensure that P_1 does not cheat, the rest of the parties of C_i send hashes of the opened $(c + v)$'s for *all* of the multiplication gates at this level to the parties of C_{i+1}. Although this introduces $\Omega(n^2)$ overhead, as long as if the width of the circuit is $\Omega(n)$, overall $O(n \cdot |C|)$ communication is still achieved. Details are as follows.

Figure 3.10: Procedure $\pi_{\text{mult-dm}}$

Usage: Multiply $[\![x]\!]_{\Delta_{i-2}}^{C_i}$ and $[\![y]\!]_{\Delta_{i-2}}^{C_i}$ held by C_i and MAC'd under C_{i-2}'s key so that C_{i+2} outputs $[\![x \cdot y]\!]_{\Delta_i}^{C_{i+2}}$ MAC'd under C_i's key.

1. All parties in C_{i-2} first agree on a special party P_{king} in C_{i-1}.

2. Then C_{i-2} and C_{i-1} invoke $\pi_{\text{get-combined-prep}}$ on input $(\text{trip}, C_{i-2}, C_{i-2} \cup C_{i-1})$ to get partially authenticated BeDOZa triple $(\langle a \rangle^{C_{i-2}, C_{i-2} \cup C_{i-1}}, \langle b \rangle^{C_{i-2}, C_{i-2} \cup C_{i-1}}, [c]^{C_{i-2}})$ and $(\text{rand}, C_{i-2}, C_{i-2} \cup C_{i-1})$ to get $\langle v \rangle^{C_{i-2}, C_{i-2} \cup C_{i-1}}$. They also locally compute $[c+v]^{C_{i-2}}$.

3. Then C_{i-2} invokes $\pi_{\text{get-x-comm-shrs}}$ on $\langle a \rangle^{C_{i-2}, C_{i-2} \cup C_{i-1}}, \langle b \rangle^{C_{i-2}, C_{i-2} \cup C_{i-1}}, \langle v \rangle^{C_{i-2}, C_{i-2} \cup C_{i-1}}$ (with C_{i+1}) to get SPDZ sharings $[\![a]\!]^{C_{i-2}}_{\Delta_{i-2}}, [\![b]\!]^{C_{i-2}}_{\Delta_{i-2}}$ ($[\![v]\!]^{C_{i-2}}_{\Delta_{i-2}}$ is not needed).

4. Next, parties in C_{i-2} in parallel invoke $\pi_{\text{eff-reshare-dm}}$ on input $[\![a]\!]^{C_{i-2}}_{\Delta_{i-2}}, [\![b]\!]^{C_{i-2}}_{\Delta_{i-2}}$ and open shares of $[c+v]^{C_{i-2}}$ to P_{king} in C_{i-1}.

5. While P_{king} distributes opened $(c+v)$ to the parties of C_i, all parties in C_{i-1} then invoke $\pi_{\text{eff-reshare-dm}}$ on input $[\![a]\!]^{C_{i-1}}_{\Delta_{i-2}}, [\![b]\!]^{C_{i-1}}_{\Delta_{i-2}}$.

6. Now, parties in C_i agree on a special party P'_{king} in C_{i+1} and then compute $[\![x+a]\!]^{C_i}_{\Delta_{i-2}} \leftarrow [\![x]\!]^{C_i}_{\Delta_{i-2}} + [\![a]\!]^{C_i}_{\Delta_{i-2}}$, and $[\![y+b]\!]^{C_i}_{\Delta_{i-2}} \leftarrow [\![y]\!]^{C_i}_{\Delta_{i-2}} + [\![b]\!]^{C_i}_{\Delta_{i-2}}$.

7. Each $P_j \in C_i$ also interprets the next pairwise shared random values from $\mathcal{F}_{\text{prep}}$, $\{s_{j,l}\}_{l \in C_{i+1}}$ as keys to a universal hash family $\mathsf{H} = \{\mathsf{H}_s : \mathbb{F}_p^M \to \mathbb{F}_p\}$ and computes $h_{j,l} \leftarrow \mathsf{H}_{s_{j,l}}(\{(c+v)_m\}_{m \in [T_i]})$ for each $P_l \in C_{i+1}$, on input the T_i values $(c+v)_m$ used for the T_i multiplications computed by this committee.

8. In parallel: (i) each P_j in C_i invokes $\pi_{\text{eff-reshare-dm}}$ on input $[\Delta_i]^{C_i}$, opens their share of $[\![x+a]\!]^{C_i}_{\Delta_{i-2}}, [\![y+b]\!]^{C_i}_{\Delta_{i-2}}$ to P'_{king} in C_{i+1}, and sends to each $P_l \in C_{i+1}$ the computed hash value $h_{j,l}$; while (ii) only $P_1 \in C_i$ sends $(c+v)$ to each $P_l \in C_{i+1}$.

9. Each P_l in C_{i+1} first uses the pairwise shared random values from $\mathcal{F}_{\text{prep}}$, $\{s_{j,l}\}_{j \in C_i}$ to compute $h'_{j,l} \leftarrow \mathsf{H}_{s_{j,l}}(\{(c+v)_m\}_{m \in [T_i]})$, checks that each $h'_{j,l} = h_{j,l}$, and aborts if any check fails; else continues.

10. Then use $[\![v]\!]^{C_{i+1}}_{\Delta_i}$ (obtained from $\pi_{\text{get-x-comm-shrs}}$) and opened $(c+v)$ to compute authenticated $[\![c]\!]^{C_{i+1}}_{\Delta_i} \leftarrow ((c+v) - [v]^{C_{i+1}}, [\Delta_i]^{C_{i+1}} \cdot (c+v) - [\Delta_i \cdot v]^{C_{i+1}}, [\Delta_i]^{C_{i+1}})$.

11. Then while P'_{king} distributes opened $d = x + a$ and $e = y + b$ to the parties of C_{i+2}, all parties in C_{i+1} invoke $\pi_{\text{eff-reshare-dm}}$ on input $[\![a]\!]^{C_{i+1}}_{\Delta_i}$, $[\![b]\!]^{C_{i+1}}_{\Delta_i}$ (also obtained from $\pi_{\text{get-x-comm-shrs}}$), and $[\![c]\!]^{C_{i+1}}_{\Delta_i}$.

12. C_{i+2} finally locally compute $[\![x \cdot y]\!]^{C_{i+2}}_{\Delta_i} \leftarrow de - d\,[\![b]\!]^{C_{i+2}}_{\Delta_i} - e\,[\![a]\!]^{C_{i+2}}_{\Delta_i} + [\![c]\!]^{C_{i+2}}_{\Delta_i}$.

13. Parties in C_{i+2} will also invoke $\pi_{\text{MAC-check-hm}}$ on input $(\text{update}, \{((x_m + a_m, [\Delta_{i-2} \cdot (x_m + a_m)]^{C_{i+2}}), (y_m + b_m, [\Delta_{i-2} \cdot (y_m + b_m)]^{C_{i+2}}))\}_{m \in [T]})$ corresponding to all of the openings of the above form they receive for the multiplication gates at this layer of the circuit.

Lemma 3.7. *Procedure $\pi_{\text{mult-dm}}$'s transcript is simulatable.*

Proof. First, we know that $\pi_{\text{get-x-comm-shrs}}$ and $\pi_{\text{eff-reshare-dm}}$ are simulatable by random values from Lemmas 3.6 and 3.2. Also, since a, b, v are uniformly random and unknown to the adversary by security of $\mathcal{F}_{\text{prep}}$, openings $(c + v), (x + a), (y + b)$ are simulatable by random values. Also, the simulator knows the universal hash keys that honest parties use to compute the hashes on the opened values (which it also knows), thus it can simulate these hashes itself. □

As with the honest majority protocol of Section 3.4 and Le Mans, we also need to account for the errors that can be introduced in the originally unauthenticated c parts of multiplication triples, that $\pi_{\text{MAC-check-dm}}$ will not catch. Indeed, as with Le Mans, $\mathcal{F}_{\text{prep}}$ also allows the adversary to add errors to c of the form $\{a^j \cdot \delta^{j,l}_b + b^j \cdot \delta^{j,l}_a\}_{j \in \mathcal{H}_{C_{i-2}}, l \in \mathcal{T}_{C_{i-2}}}$, where a^j, b^j are the honest parties' shares of the a and b parts of the triple, and $\delta^{j,l}_b, \delta^{j,l}_a$ are chosen by the adversary. These errors could cause multiplications to be computed incorrectly. We thus use similar ideas to [Rachuri and Scholl 2022; Choudhuri et al. 2021], with ideas rooted in [Chida et al. 2018], to compute a randomized version of the circuit that will be used to verify multiplications. The details are in Procedure $\pi_{\text{mult-verify-dm}}$ below. Note that in order to "keep up" with the invariant that we used in $\pi_{\text{mult-dm}}$, we need to use similar techniques to ensure that the accumulators used in $\pi_{\text{mult-verify-dm}}$ are MAC'd under the same keys as the multiplication gate outputs.

Figure 3.11: Procedure $\pi_{\text{mult-verify-dm}}$

Usage: Each committee C_i that gets the output wires of the multiplication gates of some layer ℓ of the circuit incrementally updates a multiplication verification state $(\llbracket u' \rrbracket_{\Delta_{i-2}}^{C_i}, \llbracket w' \rrbracket_{\Delta_{i-2}}^{C_i})$, which the final committees at the end of the computation use to check that all multiplications throughout the protocol were performed correctly.

Init: Each Party P_i in committee C_5 (the first to get output from $\pi_{\text{mult-dm}}$) initially defines their shares of $\llbracket u' \rrbracket_{\Delta_5}^{C_7}, \llbracket w' \rrbracket_{\Delta_5}^{C_7}$ as $(u')^i = (w')^i \leftarrow 0$ (same for the SPDZ MAC shares).

Update State: On input $(\text{update}, \{(\llbracket z_m \rrbracket_{\Delta_{i-2}}^{C_i}, \llbracket rz_m \rrbracket_{\Delta_{i-2}}^{C_i}\}_{m \in [T]})$ from committee C_i, where $\{(\llbracket z_m \rrbracket_{\Delta_{i-2}}^{C_i}, \llbracket rz_m \rrbracket_{\Delta_{i-2}}^{C_i}\}_{m \in [T]}$ were the output wires of multiplication gates computed by C_i:

1. C_{i-4} and C_{i-3} invoke $\pi_{\text{get-combined-prep}}$ with $(\text{rand}, C_{i-4}, C_{i-4} \cup C_{i-3})$ twice to get $\langle s \rangle^{C_{i-4}, C_{i-4} \cup C_{i-3}}, \langle s' \rangle^{C_{i-4}, C_{i-4} \cup C_{i-3}}$ and then C_{i-4} invokes $\pi_{\text{get-x-comm-shrs}}$ on them (with C_{i-1}) so that C_{i-4} gets $\llbracket s \rrbracket_{\Delta_{i-4}}^{C_{i-4}}, \llbracket s' \rrbracket_{\Delta_{i-4}}^{C_{i-4}}$.

2. Then C_{i-4} invokes $\pi_{\text{eff-reshare-dm}}$ on $\llbracket s \rrbracket_{\Delta_{i-4}}^{C_{i-4}}, \llbracket s' \rrbracket_{\Delta_{i-4}}^{C_{i-4}}$ and then C_{i-3} does the same.

3. Now, C_{i-2} first agrees on P_{king} in C_{i-1} then in parallel: (i) invokes $\pi_{\text{eff-reshare-dm}}$ on $[\Delta_{i-2}]^{C_{i-2}}$; and (ii) opens shares of $\llbracket u+s \rrbracket_{\Delta_{i-4}}^{C_{i-2}}, \llbracket w+s' \rrbracket_{\Delta_{i-4}}^{C_{-2i}}$ to P_{king}.

4. Committee C_{i-1} invokes $\pi_{\text{get-combined-prep}}$ with $(\text{rand}, C_{i-1}, C_i)$ so that $P_j \in C_{i-1}$ gets $(\alpha^j, \{M^{j,l}\}_{l \in C_i})$ and $P_l \in C_i$ gets $(\Delta^l, \{K^{l,j}\}_{j \in C_{i-1}})$.

5. While P_{king} distributes opened $(u+s), (w+s')$ to the parties of C_i, all parties in C_{i-1} invoke $\pi_{\text{eff-reshare-dm}}$ on $\llbracket s \rrbracket_{\Delta_{i-2}}^{C_{i-1}}, \llbracket s' \rrbracket_{\Delta_{i-2}}^{C_{i-1}}$ (obtained from $\pi_{\text{get-x-comm-shrs}}$) and send to each $P_l \in C_i$ their share α^j and corresponding MAC for P_l, $M^{j,l}$.

6. Parties P_l in C_i then locally compute $\llbracket u \rrbracket_{\Delta_{i-2}}^{C_i} \leftarrow (u+s) - \llbracket s \rrbracket_{\Delta_{i-2}}^{C_i}$ and $\llbracket w \rrbracket_{\Delta_{i-2}}^{C_i} \leftarrow (w+s') - \llbracket s' \rrbracket_{\Delta_{i-2}}^{C_i}$.

7. Then each $P_l \in C_i$ locally checks that $M^{j,l} = \alpha^j \cdot \Delta^l + K^{l,j}$, for each $P_j \in C_{i-1}$, and aborts if any fail. If not, let $\alpha = \sum_{j \in C_{i-1}} \alpha^j$.

8. Finally, each $P_l \in C_i$ locally computes $[\![u]\!]^{C_i}_{\Delta_{i-2}} \leftarrow [\![u]\!]^{C_i}_{\Delta_{i-2}} + \sum_{m=1}^{T}(\alpha)^m \cdot [\![rz_m]\!]^{C_i}_{\Delta_{i-2}}$ and $[\![w]\!]^{C_i}_{\Delta_{i-2}} \leftarrow [\![w]\!]^{C_i}_{\Delta_{i-2}} + \sum_{m=1}^{T}(\alpha)^m \cdot [\![z_m]\!]^{C_i}_{\Delta_{i-2}}$ (here $(\alpha)^m$ is the m-th power of α).

Check State: On input check from the clients C_{clnt}:

1. The clients C_{clnt} first open $[\![r]\!]^{C_{\mathsf{clnt}}}_{\Delta_1}$ and check its MAC by running both phases of $\pi_{\text{MAC-check-dm}}$.

2. Then they all open $([\![u]\!]^{C_{\mathsf{clnt}}}_{\Delta_{\ell-1}} - r \cdot [\![w]\!]^{C_{\mathsf{clnt}}}_{\Delta_{\ell-1}})$ and check its MAC by running both phases of $\pi_{\text{MAC-check-dm}}$. If the opened value is 0, output Accept; else Reject.

Lemma 3.8. *Procedure $\pi_{\text{mult-verify-dm}}$ is correct, i.e., it accepts if all multiplications are computed correctly. Moreover, it is sound, i.e., it rejects except with probability at most $(2 + \max_i T_i)/p$ in case at least one multiplication is not correctly computed. Furthermore, the transcript of **Update State** is simulatable.*

Proof. For soundness, we consider all of the points in which the adversary can inject error, either when multiplying $[\![r]\!]^{C_3}_{\Delta_1}$ with each input $[\![v_j]\!]^{C_3}_{\Delta_1}$ or when a given committee C_i is updating $[\![u]\!]^{C_i}_{\Delta_{i-2}}$ and $[\![w]\!]^{C_i}_{\Delta_{i-2}}$ based on multiplication gate outputs it has received. First, note that with all-but-negligible probability, the additive error for some $[\![c]\!]^{C_i}_{\Delta_{i-2}}$ in a multiplication triple is independent of the opened $(x + a), (y + b)$ for that multiplication. This is because the (potentially incorrect) $(c + v)$ that is opened to some honest party $P_j \in C_{i-2}$ before $(x + a), (y + b)$ are opened is the same as that received by an honest party $P_l \in C_{i-1}$ with all-but-negligible probability. We know this because their shared universal hash key $s_{j,l}$ from $\mathcal{F}_{\text{prep}}$ is uniformly random and unknown to the adversary. So, since H is a universal hash family, it holds that if P_j gets $\{(c + v)_m\}_{m\in[T_i]}$ and P_l gets a different $\{(c + v)'_m\}_{m\in[T_i]}$,

$$\Pr[\mathsf{H}_{s_{j,l}}(\{(c + v)_m\}_{m\in[T_i]}) = h_{j,l} = h'_{j,l} = \mathsf{H}_{s_{j,l}}(\{(c + v)'_m\}_{m\in[T_i]})] \leq 1/p.$$

So, since $(c + v)$ is opened before $(x + a)$ and $(y + b)$ and v is uniformly random and independent of them, the additive error for $[\![c]\!]^{C_i}_{\Delta_{i-2}}$ must be independent of them (along with the challenge α which is opened even later).

The adversary can thus only inject the following kind of errors:

1. The $[\![c_m]\!]^{C_i}_{\Delta_{i-2}}$ part of the m-th multiplication triple $([\![a_m]\!]^{C_i}_{\Delta_{i-2}}, [\![b_m]\!]^{C_i}_{\Delta_{i-2}}, [\![c_m]\!]^{C_i}_{\Delta_{i-2}})$ used to compute $[\![x_m y_m]\!]^{C_i}_{\Delta_{i-2}}$ may have errors $\{a^j_m \cdot \delta^{j,l}_{b_m} + b^j_m \cdot \delta^{j,l}_{a_m}\}_{j \in \mathcal{H}_{C_{i-4}}, l \in \mathcal{T}_{C_{i-4}}} + \delta_{c_m}$. The errors in the brackets come from adversarial action in $\mathcal{F}_{\mathsf{prep}}$ while the δ_c comes from the fact that c is not authenticated, so the adversary can insert more error when resharing/authenticating them. Call this error $\varepsilon_{i,m}$. (Note this is the only source of error when computing some $[\![rv_j]\!]^{C_5}_{\Delta_3}$.)

2. Additionally, the $[\![c'_m]\!]^{C_i}_{\Delta_{i-2}}$ part of the multiplication triple $([\![a'_m]\!]^{C_i}_{\Delta_{i-2}}, [\![b'_m]\!]^{C_i}_{\Delta_{i-2}}, [\![c'_m]\!]^{C_i}_{\Delta_{i-2}})$ used to compute $[\![r x_m y_m]\!]^{C_i}_{\Delta_{i-2}}$ may have the same kind of errors:

$$\{(a'_m)^j \cdot (\delta')^{j,l}_{b_m} + (b'_m)^j \cdot (\delta')^{j,l}_{a_m}\}_{j \in \mathcal{H}_{C_{i-4}}, l \in \mathcal{T}_{C_{i-4}}} + \delta_{c_m}.$$

   Call this error $\varepsilon'_{i,m}$.

3. Also, we must consider the accumulated error $\eta_{i,m}$ on the randomized value $r x_m$ from previous gates.

So, at the end of the computation (after d multiplications), if the check passes we will have:

$$0 = [\![u]\!]^{C_{\mathsf{clnt}}}_{\Delta_{\ell-1}} - r \cdot [\![w]\!]^{C_{\mathsf{clnt}}}_{\Delta_{\ell-1}}$$

$$= \sum_{j=1}^{T_0} \alpha_0^j \cdot (rv_j + \varepsilon_{0,j}) + \sum_{i=1}^{d} \sum_{m=1}^{T_i} \alpha_i^m \cdot ((r x_m + \eta_{i,m}) \cdot y_m + \varepsilon'_{i,m}) -$$

$$r \cdot \left(\sum_{i=1}^{d} \sum_{m=1}^{T_i} \alpha_i^m \cdot (x_m \cdot y_m + \varepsilon_{i,m}) + \sum_{j=1}^{M} \alpha_0^j \cdot v_j \right)$$

$$= \sum_{i=1}^{d} \sum_{m=1}^{T_i} \alpha_i^m (\eta_{i,m} \cdot y_m + \varepsilon'_{i,m} - r \cdot \varepsilon_{i,m}) + \sum_{j=1}^{M} \alpha_0^j \cdot \varepsilon_{0,j}.$$

Now, note that the corrupt parties of committee C_{i-1} cannot forge their share α^j of α to any of the honest parties of C_i except with probability $1/p$, by the security of the information-theoretic MAC provided by $\mathcal{F}_{\mathsf{prep}}$. Thus, the reconstructed challenge α must indeed be uniformly random and independent of all other values. We analyze the two following cases:

Case 1: *There is some j such that $\varepsilon_{0,j} \neq 0$.* In this case, it is clear that the above polynomial is non-zero. Therefore, since each α_i is unknown to the adversary and sampled uniformly at random and independently of all other values, the Schwartz-Zippel Lemma tells us that the evaluation of this polynomial on these α_i equals 0 with probability at most $\max_i T_i/p$.

Case 2: *For all j, $\varepsilon_{0,j} = 0$.* Let layer i^* be the first in which the adversary injected error into a multiplication; i.e., $\varepsilon'_{i^*,m} \neq 0$ and/or $\varepsilon_{i^*,m} \neq 0$ for some m. Note that since this is the first such layer, it must be that $\eta_{i,m} = 0$ for all i^*, m. So,

$$0 = \sum_{i=1}^{d} \sum_{m=1}^{T_i} \alpha_i^m (\eta_i \cdot y_m + \varepsilon'_i - r \cdot \varepsilon_i)$$

$$= \sum_{m=1}^{T_{i^*}} \alpha_{i^*}^m (\varepsilon'_{i^*,m} - r \cdot \varepsilon_{i^*,m}) + \sum_{i \in [d] \setminus \{i^*\}} \sum_{m=1}^{T_i} \alpha_i^m (\eta_i \cdot y_m + \varepsilon'_i - r \cdot \varepsilon_i).$$

First, for the given m where the adversary injected error, $(\varepsilon'_{i^*,m} - r \cdot \varepsilon_{i^*,m}) = 0$ can only happen with probability $1/p$ since r is unknown to the adversary and sampled uniformly and independently of all other values. Now, if $(\varepsilon'_{i^*,m} - r \cdot \varepsilon_{i^*,m}) \neq 0$, then the above polynomial is non-zero. Since each α_i is unknown to the adversary and sampled uniformly at random and independently of all other values, the Schwartz-Zippel Lemma tells us that the evaluation of this polynomial on these α_i equals 0 with probability at most $\max_i T_i/p$.

Thus, the total probability that the adversary can inject some error is upper bounded by $(2 + \max_i T_i)/p$.

Correctness clearly holds if all errors are 0. Finally, From Lemma 3.6 we know that $\pi_{\text{get-x-comm-shrs}}$ is simulatable, and from Lemma 3.2, we know that $\pi_{\text{eff-reshare-dm}}$ is simulatable. Also, since s, s' are uniformly random and unknown to the adversary by the security of $\mathcal{F}_{\text{prep}}$, openings $(u+s)$, $(w+s')$ are simulatable by random values. Additionally, each α^j is sampled uniformly at random in $\mathcal{F}_{\text{prep}}$, so the simulator can simulate these, and their corresponding MACs by using the MAC keys obtained from $\mathcal{F}_{\text{prep}}$. Finally, from Lemma 3.4, we know that $\pi_{\text{MAC-check-dm}}$'s transcript is indeed simulatable. □

3.2.2.5 Dishonest Majority Main Protocol

With all of the previous tools in place, we can finally present our full-fledged actively secure, dishonest majority MPC protocol in the fluid setting, achieving linear communication complexity and maximal fluidity. The clients first use $\pi_{\text{eff-key-switch}}$ to securely transfer authenticated versions of their inputs to C_2. The committees then proceed to compute both the regular and randomized version of the circuit on the authenticated inputs, using $\pi_{\text{mult-dm}}$ as well as addition and identity gate procedures that work similarly using the same "mask, open to king, and unmask" paradigm along with some local computation. Addition and identity gates also need to preserve the invariant on MACs discussed in Section 3.2.2.4. The committees also make sure to update the accumulators of $\pi_{\text{MAC-check-dm}}$ and $\pi_{\text{mult-verify-dm}}$ along the way with each opening and multiplication, respectively. We note that, unlike the honest majority protocol, the outputs of the final circuit layer will be shared by the clients themselves. Once all circuit layers have been computed, the clients invoke the **Check State** phases of $\pi_{\text{MAC-check-dm}}$ and $\pi_{\text{mult-verify-dm}}$, then reconstruct the outputs. We note that, as is remarked in the protocols of [Rachuri and Scholl 2022; Choudhuri et al. 2021], if the clients indeed have access to a broadcast channel in the last round of the protocol, or implement a broadcast over their point-to-point channels, then security with unanimous abort is achieved by having the clients broadcast "abort", if their check on their output fails.

Figure 3.12: Protocol $\Pi_{\text{main-dm}}$

Preprocessing Phase: All parties $P_i \in \mathcal{U}$ invoke $\mathcal{F}_{\text{prep}}$ to receive their share of the global MAC key Δ^i, along with enough pairwise random sharings, multiplication triples, and sharings of 0.

Input Phase: To form a SPDZ sharing of an input x_i possessed by $P_i \in C_{\text{clnt}}$:

1. P_i invokes $\pi_{\text{eff-key-switch}}$ on $(x_i, \Delta^i \cdot x_i, \Delta^i)$ with C_2 (through C_1).
2. C_1 invokes $\pi_{\text{get-combined-prep}}$ with (rand, C_1, C_1) to get $\langle r \rangle^{C_1, C_1}$ then invokes π_{convert} on it to get SPDZ sharing $[\![r]\!]^{C_1}_{\Delta_1}$, and finally $\pi_{\text{eff-reshare-dm}}$ on this.
3. C_2 invokes $\pi_{\text{eff-reshare-dm}}$ on $[\![x]\!]^{C_2}_{\Delta_1}$ (from $\pi_{\text{eff-key-switch}}$ above) and $[\![r]\!]^{C_2}_{\Delta_1}$.
4. Finally, C_3 invokes $\pi_{\text{mult-dm}}$ on $[\![x]\!]^{C_3}_{\Delta_1}$ and $[\![r]\!]^{C_3}_{\Delta_1}$, as well as the identity gate procedure (below) on $[\![x]\!]^{C_3}_{\Delta_1}$ so that C_5 gets $[\![x]\!]^{C_5}_{\Delta_3}$ and $[\![x \cdot r]\!]^{C_5}_{\Delta_3}$.
5. Finally, C_5 invokes $\pi_{\text{mult-verify-dm}}$ on input $(\text{update}, \{([\![x_i]\!]^{C_5}_{\Delta_3}, [\![rx_i]\!]^{C_5}_{\Delta_3})\}_{i \in [|C_{\text{clnt}}|]})$, corresponding to each input.

Execution Phase:

1. Each committee C_i of the execution phase first invokes $\pi_{\text{eff-reshare-dm}}$ on $[\![r]\!]^{C_i}_{\Delta_1}$.
2. In parallel, every other committee (with the help of the others) will compute the gates at each layer of the circuit as below:

Addition: To perform addition on $[\![x]\!]^{C_i}_{\Delta_{i-2}}$ and $[\![y]\!]^{C_i}_{\Delta_{i-2}}$ (and identically for $[\![rx]\!]^{C_i}_{\Delta_{i-2}}$ and $[\![ry]\!]^{C_i}_{\Delta_{i-2}}$):

1. C_{i-2} and C_{i-1} invoke $\pi_{\text{get-combined-prep}}$ with $(\text{rand}, C_{i-2}, C_{i-2} \cup C_{i-1})$ to get $\langle s \rangle^{C_{i-2}, C_{i-2} \cup C_{i-1}}$ and then invokes $\pi_{\text{get-x-comm-shrs}}$ on it (with C_{i+1}) to get SDPZ sharing $[\![s]\!]^{C_{i-2}}_{\Delta_{i-2}}$.
2. Then C_{i-2} invokes $\pi_{\text{eff-reshare-dm}}$ on $[\![s]\!]^{C_{i-2}}_{\Delta_{i-2}}$ and C_{i-1} does the same.
3. Now, C_i first agrees on P_{king} in C_{i+1} then locally computes $[\![x + y]\!]^{C_i}_{\Delta_{i-2}}$.

4. Then C_i in parallel: (i) invokes $\pi_{\text{eff-reshare-dm}}$ on $[\Delta_i]^{C_i}$; and (ii) opens shares of $[\![x+y+s]\!]^{C_i}_{\Delta_{i-2}}$ to P_{king}.

5. While P_{king} distributes opened $(x+y+s)$ to the parties of C_{i+2}, all parties of C_{i+1} invoke $\pi_{\text{eff-reshare-dm}}$ on $[\![s]\!]^{C_{i+1}}_{\Delta_i}$ (obtained through $\pi_{\text{get-x-comm-shrs}}$).

6. Parties in C_{i+2} finally locally compute $[\![x+y]\!]^{C_{i+2}}_{\Delta_i} \leftarrow (x+y+s) - [\![s]\!]^{C_{i+2}}_{\Delta_i}$.

7. Parties in C_{i+2} will also invoke $\pi_{\text{MAC-check-hm}}$ on input $(\text{update}, \{(x_m + y_m + s_m, [\Delta_{i-2} \cdot (x_m + y_m + s_m)]^{C_{i+2}})\}_{m \in [T]})$ corresponding to all of the openings of the above form they receive for the addition gates at this layer of the circuit.[a]

Identity Gates: C_i forwards $[\![x]\!]^{C_i}_{\Delta_{i-2}}, [\![rx]\!]^{C_i}_{\Delta_{i-2}}$ to C_{i+2} (so that they are MAC'd under Δ_i) in a similar fashion as addition above.

Multiplication: To multiply $[\![x]\!]^{C_i}_{\Delta_{i-2}}$ and $[\![y]\!]^{C_i}_{\Delta_{i-2}}$, invoke $\pi_{\text{mult-dm}}$ on them (and identically for $[\![rx]\!]^{C_i}_{\Delta_{i-2}}$ and $[\![y]\!]^{C_i}_{\Delta_{i-2}}$). Then invoke $\pi_{\text{mult-verify-hm}}$ on input $(\text{update}, \{([\![x_m y_m]\!]^{C_{i+2}}_{\Delta_i}, [\![(rx)_m y_m]\!]^{C_{i+2}}_{\Delta_i})\}_{m \in [T_i]})$, corresponding to each multiplication performed at this layer of the circuit.

Output Phase:

1. Clients in C_{clnt} first invoke $\pi_{\text{MAC-check-dm}}$ on check. If it outputs Reject, then abort; else, continue. This takes 2 rounds.

2. Clients in C_{clnt} then invoke $\pi_{\text{mult-verify-dm}}$ on check. If it outputs Reject, then abort; else, continue. This takes 6 more rounds.

3. Clients finally open each output wire $[\![z]\!]^{C_{\text{clnt}}}_{\Delta_{\ell-1}}$ and check their MACs by running both phases of $\pi_{\text{MAC-check-dm}}$. If it outputs Reject, then abort; else, output each z. This takes 3 more rounds.

[a]This invocation can be combined with that of the multiplication gates for this circuit layer.

Theorem 3.9. *Let $\mathcal{A}$ be an* R*-adaptive adversary in* $\Pi_{\text{main-dm}}$. *Then the protocol UC-securely computes* $\mathcal{F}_{\text{DABB}}$ *in the presence of* $\mathcal{A}$ *in the* $(\mathcal{F}_{\text{prep}}, \mathcal{F}_{\text{commit}})$*-hybrid model.*

Proof. We construct a Simulator ($\mathcal{S}$) that runs the adversary ($\mathcal{A}$) as a subroutine, and is given access to $\mathcal{F}_{\mathsf{DABB}}$. It internally emulates the functionalities $\mathcal{F}_{\mathsf{prep}}$ and $\mathcal{F}_{\mathsf{commit}}$ and we implicitly assume that it passes all communication between $\mathcal{A}$ and the environment $\mathcal{Z}$. It keeps track of the current committee via inputs (Init, C) and $(\mathsf{Next\text{-}Committee}, C)$ from $\mathcal{F}_{\mathsf{DABB}}$ (and therefore in which committees to simulate corresponding communication for circuit gates). The simulator uses bad, initially set to 0, to detect any bad behavior from $\mathcal{A}$. If so, it sets bad $\leftarrow 1$. The simulation proceeds as follows:

Init: On input (Init, C), keep track of $\mathcal{A}$'s inputs to $\mathcal{F}_{\mathsf{prep}}$, including any additive errors $\delta_a \neq 0$ or $\delta_b \neq 0$ for any pairwise multiplication triples (this might not be an issue yet, as long as if the eventual additive error on any c of any triple ends up as 0; see below).

Input: On input $(\mathsf{Input}, \mathsf{id}_x)$ (for both honest and adversarial inputs), simulate $\pi_{\text{eff-key-switch}}$ as in Lemma 3.5, π_{convert} as in Lemma 3.3, $\pi_{\text{eff-reshare-dm}}$ as in Lemma 3.2, and $\pi_{\text{MAC-check-dm}}$ as in Lemma 3.4. If $\mathcal{A}$ cheats when sending some universal hash value during $\pi_{\text{MAC-check-dm}}$, abort. If $\mathcal{A}$ cheats either when resharing or opening a value (i.e., by sending a wrong share), set bad $\leftarrow 1$. Also, simulate the multiplication as below.

Addition (and similarly for identity gates): On input $(\mathsf{Add}, \mathsf{id}_z, \mathsf{id}_x, \mathsf{id}_y)$, simulate $\pi_{\text{get-x-comm-shrs}}$ as in Lemma 3.6, $\pi_{\text{eff-reshare-dm}}$ as in Lemma 3.2, and $\pi_{\text{MAC-check-dm}}$ as in Lemma 3.4. Additionally, simulate the opening of $[\![x + y + s]\!]^{C_i}_{\Delta_{i-2}}$ with random values. Since s is uniformly random and unknown to $\mathcal{A}$, this is a perfect simulation. If $\mathcal{A}$ cheats when sending some universal hash value during $\pi_{\text{MAC-check-dm}}$, abort. If $\mathcal{A}$ cheats either when resharing or opening a value (i.e., by sending a wrong share), set bad $\leftarrow 1$.

Multiplication: On input $(\mathsf{Mult}, \mathsf{id}_z, \mathsf{id}_x, \mathsf{id}_y)$, simulate $\pi_{\text{mult-dm}}$ as in Lemma 3.7 and $\pi_{\text{mult-verify-dm}}$ as in Lemma 3.8. If $\mathcal{A}$ cheats when sending some universal hash value during $\pi_{\text{MAC-check-dm}}$ or $\pi_{\text{mult-dm}}$, abort. If for any c part of a multiplication triple, the additive error $\delta_c + \sum_{j \in \mathcal{H}_{C_{i-2}}, l \in \mathcal{T}_{C_{i-2}}} a^j \cdot \delta_b^{j,l} + b^j \cdot \delta_a^{j,l} \neq 0$, set bad $\leftarrow 1$. Additionally, if $\mathcal{A}$ cheats either when resharing or opening a value (i.e., by sending a wrong share), set bad $\leftarrow 1$.

Output: If $\mathcal{S}$ ever set bad $\leftarrow 1$ because $\mathcal{A}$ cheated when opening or resharing a value, $\mathcal{S}$ sends random values for σ on behalf of the honest parties, then aborts. Otherwise, $\mathcal{S}$ records $\{\sigma^i\}_{i \in \mathcal{T}_{C_{\mathsf{clnt}}}}$ sent to $\mathcal{F}_{\mathsf{commit}}$ by $\mathcal{A}$ in the check state phase of $\pi_{\mathsf{MAC\text{-}check\text{-}dm}}$, samples random shares for the honest parties such that $\sum_{i \in C_{\mathsf{clnt}}} \sigma^i = 0$ and sends them to $\mathcal{A}$. If $\mathcal{A}$ cheats during the *check state* phase of $\pi_{\mathsf{MAC\text{-}check\text{-}dm}}$ (e.g., by committing to the wrong MAC check value σ^i), $\mathcal{S}$ aborts. In the *check state* phase of $\pi_{\mathsf{mult\text{-}verify\text{-}dm}}$, $\mathcal{S}$ sends random shares on behalf of the honest parties for the opening of r. If $\mathcal{A}$ cheats by opening the wrong values for r, $\mathcal{S}$ aborts after the MAC check (as above). If $\mathcal{S}$ ever set bad $\leftarrow 1$ because $\mathcal{A}$ added non-zero error to the c part of a multiplication triple, $\mathcal{S}$ sends random values on behalf of the honest parties for $(u - r \cdot w)$, then aborts. Otherwise, $\mathcal{S}$ records the values sent by $\mathcal{A}$ for $(u - r \cdot w)$, then samples shares such that $(u - r \cdot w) = 0$, and sends them to $\mathcal{A}$. If $\mathcal{A}$ cheats when opening $(u - r \cdot w)$, $\mathcal{S}$ aborts after the MAC check (as above).

Finally, $\mathcal{S}$ gets the outputs from $\mathcal{F}_{\mathsf{DABB}}$ and forwards it to $\mathcal{A}$. $\mathcal{S}$ then forwards whatever it receives from $\mathcal{A}$ back to $\mathcal{F}_{\mathsf{DABB}}$.

From all of the Lemmas, we have that the simulation is perfect up until the output phase. By Lemmas 3.4 and 3.8, $\mathcal{A}$ is only able to cheat in the real world with probability negligible in p. Thus, the distance between the real-world and the simulation is negligible in p. □

3.3 Dishonest Majority Preprocessing Size is Tight

3.3.1 Technical Overview

Now we provide an overview for the lower bound on the amount of preprocessed data, for the dishonest majority case. We do this in the context of secure message transmission (SMT). Assume we have some sender A who wants to send some secret value x to a receiver B through two committees that are not known ahead of time. This is related to fluid MPC: we can think of an identity function that is to be computed using two committees. That said, assume that between A and B, there are two (non-overlapping) committees C_1 and C_2, each of size n, that are chosen at random from the larger universe $\mathcal{U}$ of parties of size N. Furthermore, assume that some adversary $\mathcal{A}$ that is trying to learn x is able to (passively) corrupt all-but-one party in each of C_1 and C_2, as well as any other parties in $\mathcal{U}$. In such a setting, we also allow for some *global preprocessing protocol* that the parties of $\mathcal{U}$ can run amongst each other *before* the secret x or committees C_1 and C_2 are chosen. We show that if the size of each preprocessing state is $o(N \cdot |x|)$, then the total communication must be $\Omega(n^2 \cdot |x|)$. The intuition is as follows.

Suppose that $\mathcal{A}$ corrupts all but the first parties of each committee. Furthermore, suppose, towards contradiction, that the size of the message $c_{1,1}$ that the first party of C_1 sends to the first party of C_2 is small ($\ll |x|$). First, this means that $\mathcal{A}$ can guess this message with high probability. Now, suppose that the preprocessing $r_{1,1}$ of the first party of C_1 is not anymore correlated with the preprocessing $r_{2,1}$ of the second party of C_2, than the preprocessing of the rest of the corrupted parties of C_1. This correlation is what the parties of C_1 (perhaps, implicitly) use to construct messages that will eventually result in correct transmission to the receiver B. In particular, any possible preprocessing $r'_{2,1}$ that has non-zero probability weight conditioned on the preprocessing of the parties of C_1, and thus by the above assumption, the corrupted parties of C_1, must enable correct transmission. So, since the first party of C_2 only uses $r_{2,1}$ along with the ciphertexts it receives to produce its message to the receiver B, $\mathcal{A}$ must be able to use a guess for $r_{2,1}$ conditioned on the preprocessing of corrupt parties of C_1 to produce a valid such message. Together with the other messages to the receiver B from the corrupted parties of C_2, $\mathcal{A}$ can reconstruct x with high probability.

So, it must in fact be that $r_{1,1}$ provides some unique information on the preprocessing of $r_{2,1}$ that the corrupted parties of C_1 do not already provide. However, it is just as likely that some other party in $\mathcal{U}$ could have been chosen to be the first party of C_1, in some other execution of the protocol. So, in fact *every* party outside of C_1 and C_2 must provide some unique information on the preprocessing of $r_{2,1}$. Since $\mathcal{A}$ can corrupt as many of these parties as it wishes, if $r_{2,1}$ is small enough (in particular, $o(N \cdot |x|)$), then $\mathcal{A}$ will eventually be able to reconstruct $r_{2,1}$ completely, guess (short) $c_{1,1}$ and thus again reconstruct x with high probability, as above. Therefore, a contradiction is reached, and the size of each (n^2 total) ciphertext $c_{i,j}$ must be $\Omega(|x|)$.

3.3.2 Formal Bound

In this section, we show that the per-party size of the preprocessing produced in our dishonest majority protocol $\Pi_{\mathsf{main\text{-}dm}}$ is tight in the following sense. *Any* protocol that uses more than one committee to compute some function must have per-party size of preprocessing proportional to N, i.e. the size of the entire server universe, $\mathcal{U}$.

To show this, we intuitively reduce the problem of MPC in the Fluid Model with more than two committees to the problem of simply resharing state securely. Indeed, for any such MPC protocol, after one committee finishes their step of the computation, they must securely reshare some sort of state to the next committee, since the next committee has no information about the current state of the computation (e.g., including the original inputs).

More formally, we show a lower bound on the per-party preprocessing size for Secure Message Transmission (SMT) with two committees. In such an SMT setting, there is a sender A who wishes to send some (possibly uniformly random) message x to a receiver B, but first must send some private representation of x through two committees, C_1 and C_2 that are not known ahead of time. Informally, this corresponds to the "resharing" argument in the Fluid MPC model above, since the transmitted x corresponds to the state that is being reshared by the first committee to the next.

3.3.2.1 Secure Message Transmission with Two Committees

Now we formally define SMT with two committees. In this definition, we will demand that a uniformly random message x of length λ, i.e., $x \leftarrow_\$ \{0,1\}^\lambda$, will be transmitted from A to B, after passing through committees C_1 and C_2. The two committees C_1 and C_2 can be arbitrarily chosen from a larger universe $\mathcal{U} = \{P_1, \ldots, P_N\}$ of size N. We will allow for a *preprocessing phase* to be performed *before* the input x and committees C_1 and C_2 are chosen. First we present the syntax:

- Algorithm $\{r_i\}_{P_i \in \mathcal{U}} \leftarrow_\$ \mathsf{SMT\text{-}Prep}(\mathcal{U})$ takes as input the set of parties in $\mathcal{U}$ and outputs a preprocessing state, r_i, for each $P_i \in \mathcal{U}$.
- The sender will use algorithm $\{A_i\}_{P_i \in C_1} \leftarrow_\$ \mathsf{SMT\text{-}A\text{-}Send}(x, C_1)$ to send messages A_i for each P_i in C_1, based on chosen $x \in \{0,1\}^\lambda$.
- Each $P_i \in C_1$ will then use $\{c_{i,j}\}_{P_j \in C_2} \leftarrow_\$ \mathsf{SMT\text{-}}C_1\mathsf{\text{-}Send}(r_i, A_i, C_2)$ to send message $c_{i,j}$ to each P_j in C_2.
- Next, each $P_j \in C_2$ will use algorithm $B_j \leftarrow_\$ \mathsf{SMT\text{-}}C_2\mathsf{\text{-}Send}(r_j, \{c_{i,j}\}_{P_i \in C_1}, C_1)$ to send message B_j to the receiver.
- Finally, the receiver will use algorithm $x \leftarrow \mathsf{SMT\text{-}B\text{-}Rcv}(\{B_j\}_{P_j \in C_2})$ to output the message x.

Since we are in the dishonest majority setting, we will consider any *unbounded* adversary $\mathcal{A}$ that corrupts all-but-one party in each committee, *only* during the online phase. That is, using the same notation as earlier in this section, the sizes of corruption sets $\mathcal{T}_{C_1}$ and $\mathcal{T}_{C_2}$ satisfy $t_1 < n_1$ and $t_2 < n_2$, respectively. Now, we are ready for the definition:

Definition 3.10 (Secure Message Transmission with Two Committees.). A *Secure Message Transmission with Two Committees* protocol Π_{SMT} is *perfectly-correct* if for any choice of committees $C_1, C_2 \subseteq \mathcal{U}$,

$$\Pr\big[x \leftarrow \mathsf{SMT\text{-}B\text{-}Rcv}(\{B_j\}_{P_j \in C_2}) : x \leftarrow_\$ \{0,1\}^\lambda, \{r_l\}_{P_l \in \mathcal{U}} \leftarrow_\$ \mathsf{SMT\text{-}Prep}(\mathcal{U}),$$

$$\{A_i\}_{P_i \in C_1} \leftarrow_\$ \mathsf{SMT\text{-}A\text{-}Send}(x, C_1), \forall P_i \in C_1, \{c_{i,j}\}_{P_j \in C_2} \leftarrow_\$ \mathsf{SMT\text{-}}C_1\mathsf{\text{-}Send}(r_i, A_i, C_2),$$

$$\forall P_j \in C_2, B_j \leftarrow_\$ \mathsf{SMT\text{-}}C_2\mathsf{\text{-}Send}(r_j, \{c_{i,j}\}_{P_i \in C_1}, C_1)\big] = 1.$$

Moreover, Π_{SMT} is *statistically-secure* if for any choice of committees $C_1, C_2 \subseteq \mathcal{U}$, and any choice of corruptions $\mathcal{T}_{C_1} \subseteq C_1, \mathcal{T}_{C_2} \subseteq C_2$ satisfying $t_1 < n_1$ and $t_2 < n_2$, respectively,

$$\Pr\big[x \leftarrow \mathcal{A}(\{(r_i, A_i)\}_{P_i \in \mathcal{T}_{C_1}}, \{(r_j, \{c_{l,j}\}_{P_l \in C_1})\}_{P_j \in \mathcal{T}_{C_2}}) : x \leftarrow_\$ \{0,1\}^\lambda,$$

$$\{r_l\}_{P_l \in \mathcal{U}} \leftarrow_\$ \mathsf{SMT\text{-}Prep}(\mathcal{U}), \{A_i\}_{P_i \in C_1} \leftarrow_\$ \mathsf{SMT\text{-}A\text{-}Send}(x, C_1),$$

$$\forall P_i \in C_1, \{c_{i,j}\}_{P_j \in C_2} \leftarrow_\$ \mathsf{SMT\text{-}}C_1\mathsf{\text{-}Send}(r_i, A_i, C_2),$$

$$\forall P_j \in C_2, B_j \leftarrow_\$ \mathsf{SMT\text{-}}C_2\mathsf{\text{-}Send}(r_j, \{c_{i,j}\}_{P_i \in C_1}, C_1)\big] \leq 2^{-\lambda}.$$

In the rest of the section, we will assume for simplicity that the two committees, C_1 and C_2, will each be of the same size $n_1 = n_2 = n$. Without loss of generality, we may refer to the two committees as $C_1 = \{P_1, \ldots, P_n\}$ and $C_2 = \{P_{n+1}, \ldots, P_{2n}\}$.

Communication Complexity. We will in part be concerned by the size of communication needed for a correct and secure SMT protocol. Towards this end, let $\mathsf{Comm} = \sum_{i \in C_1, j \in C_2} |c_{i,j}|$ be the total communication from some particular execution of an SMT protocol Π_{SMT}.

3.3.2.2 Lower Bound on Per-Party Preprocessing for Linear SMT

We will now prove the following lower bound which informally states that in order for an SMT protocol Π_{SMT} to have, in expectation over the choice of committees C_1, C_2 and any randomness of the algorithms, $o(n^2 \cdot \lambda)$ total communication Comm, the expected size of each preprocessing state must be $\Omega(N \cdot \lambda)$:

Theorem 3.11. *For any perfectly-secure SMT protocol Π_{SMT} for two committees C_1, C_2 of size n, if C_1, C_2 are sampled uniformly at random from the universe $\mathcal{U}$ of size N, such that $C_1 \cap C_2 \neq \emptyset$, and $\mathbb{E}_{P_i \in \mathcal{U}}[|r_i|] \leq (N - 2n + 1) \cdot \lambda/8$, then $\mathbb{E}_{C_1, C_2}[\mathsf{Comm}] \geq n^2 \cdot \lambda/4$.*

First, we provide the following lemma that will help us in proving the above theorem. Assume w.l.o.g. that the smallest ciphertext in a given execution is $c_{1,n+1}$. Also, assume that the adversary corrupts (at least) every party in the two committees except for P_1 and P_{n+1}. In particular, this gives the adversary preprocessing states $R = r_2, \ldots, r_n, r_{n+2}, \ldots, r_{2n}$, ciphertexts $\{c_{i,n+1}\}_{i \in [2,n]}$ sent by the corrupted parties of C_1 to P_{n+1}, and ciphertexts $B_{n+2}, \ldots, B_{2n}$ sent to the receiver by the corrupted parties of C_2. So, the adversary is just missing the message B_{n+1} used by the receiver B in the protocol to reconstruct x. To produce this message B_{n+1}, the adversary in addition to ciphertexts $\{c_{i,n+1}\}_{i \in [2,n]}$ it has, only needs to learn $c_{1,n+1}$ and r_{n+1}. What this Lemma intuitively shows is that if $|c_{1,n+1}| < \lambda/2$ (so that the adversary can guess it with high enough probability), then the preprocessing r_1 *must* provide some additional, non-trivial correlation with r_{n+1} that the corrupted preprocessing states, R, do not provide on their own. If this were not the case, then the adversary could simply sample r_{n+1} conditioned on R. This preprocessing would then be "close enough" to what the correlations in the entire protocol execution indicate it should be so that, by correctness, it should also work, together with the ciphertexts $\{c_{i,n+1}\}_{i \in [n]}$, to produce the missing B_{n+1}.

In the following, we use notation R_J to represent the set of preprocessing states $\{r_j\}_{j \in J}$, where $J \subseteq [N]$.

Lemma 3.12. *Assume that the two committees C_1, C_2, each of size n, are chosen uniformly at random from the universe $\mathcal{U}$ of size N, such that $C_1 \cap C_2 = \emptyset$. Also, let J be some random (fixed-size) subset of $[N]$ such that $|J| \geq 2n - 2$. If $\Pr_{C_1,C_2,i,j}[|c_{i,j}| < \lambda/2] > 1/2$, then for random $i' \neq j' \in \mathcal{U} \setminus J$, we have: $\mathbb{E}_{J,i',j'}[\mathrm{I}(r_{j'}|R_J; r_{i'})] \geq \lambda/4$.*

Proof. We start with the following inequality, where the randomness is over the choice of $J \cup \{i', j'\} \subseteq [N]$, as well as the actual generated preprocessing state $r_{i'}$:

$$\mathbb{E}_{J,i',j',r_{i'}}[\mathrm{SD}((r_{j'}|R_J \cup \{r_{i'}\}), (r_{j'}|R_J))] \leq$$

$$\mathbb{E}_{J,i',j' r_{i'}}\left[1 - \frac{1}{2}\exp(-D_{\mathrm{KL}}((r_{j'}|R_J \cup \{r_{i'}\})||(r_{j'}|R_J)))\right] \leq$$

$$1 - \frac{1}{2}\exp(-\mathbb{E}_{J,i',j',r_{i'}}[D_{\mathrm{KL}}((r_{j'}|R_J \cup \{r_{i'}\})||(r_{j'}|R_J))]) =$$

$$1 - \frac{1}{2}\exp(-\mathbb{E}_{J,i',j'}[\mathrm{I}(r_{j'}|R_J; r_{i'})]).$$

The first inequality follows from the Bretagnolle-Huber inequality [Bretagnolle and Huber 1978], and the second inequality from Jensen's inequality, since $f(x) = e^{-x}$ is convex, while the last equality is a well-known identity.

Thus, if we assume towards contradiction that $\mathbb{E}_{J,i',j'}[\mathrm{I}(r_{j'}|R_J; r_{i'})] < \lambda/4$, this means that

$$\mathbb{E}_{J,i',j',r_{i'}}[\mathsf{SD}((r_{j'}|R_J), (r_{j'}|R_J \cup \{r_{i'}\}))] < 1 - \frac{1}{2}\exp(-\lambda/4).$$

Now, it could be the case that some $2n-2$ randomly sampled indices in J correspond exactly to the first $n-1$ parties of C_1 and C_2 in a given protocol execution, and further that $P_{i'}$ is the last party chosen for C_1, and $P_{j'}$ is the last party chosen for C_2. Also, from correctness, we know that for any $r_{j'}$ that is produced by the preprocessing phase for $P_{j'}$ with non-zero probability, the SMT protocol must successfully transmit the secret x. Based on this and the above inequality, we describe the following attack: The adversary $\mathcal{A}$ corrupts the set of preprocessing states R_J and then samples guess $r'_{j'}$ for $r_{j'}$ conditioned on the states in R_j, and samples (uniformly) guess $c'_{i',j'}$ for $c_{i',j'}$. Using the guessed $r'_{j'}$ and $c'_{i,j'}$ along with the learned $\{c_{i,j'}\}_{i\in[n]\setminus\{i'\}}$ via corruptions, invoke SMT-C_2-Send to produce $B_{j'}$. Finally, using $\{B_j\}_{j\in[n+1,2n]}$, invoke SMT-B-Rcv to produce x.

Now, let us analyze the success probability of this attack. First, we have from $\Pr_{C_1,C_2,i,j}[|c_{i,j}| < \lambda/2] > 1/2$, that $\Pr[c'_{j',j} = c_{j',j}] > 2^{-\lambda/2-1}$. Next, consider the event in which $\mathcal{A}$ samples some $r'_{j'}$ conditioned on R_J that has weight-0 in the distribution of $r_{j'}$ conditioned on $R_J \cup \{r_{i'}\}$. We call such an $r'_{j'}$ a *bad* sample. From the above inequality, such an $r'_{j'}$ is sampled with probability less than $1 - \frac{1}{2}\exp(-\lambda/4)$, in expectation. In particular, this means that in expectation, $\mathcal{A}$ samples a *good* $r'_{j'}$ with probability at least $\frac{1}{2}\exp(-\lambda/4)$. Such an $r'_{j'}$ is *good* because by correctness, the protocol must successfully transmit the secret x if in fact $r'_{j'}$ were the actual preprocessing of $P_{j'}$.

Therefore, in expectation over the choice of committee members and the preprocessing $r_{i'}$, the attack by $\mathcal{A}$ succeeds with probability greater than $1/2^{\lambda}$. □

Now we can prove Theorem 3.11 using Lemma 3.12. The intuition stems from the fact that the protocol does not *a priori* know which parties will be in the committees. So, we can use Lemma 3.12 to show that in fact *many* parties outside of the two committees must in expectation provide some unique correlation with r_{n+1} (the receiver of small message $c_{1,n+1}$), in case they were actually the first party of C_1 sending this small ciphertext. As a result, if the state r_{n+1} is small enough, we can completely recover it, guess $c_{1,n+1}$, then recover x with high enough probability.

Proof of Theorem 3.11. Assume towards contradiction that $\Pr_{C_1,C_2,i,j}[|c_{i,j}| < \lambda/2] > 1/2$. Also assume that some adversary $\mathcal{A}$ in a given execution of some Π_{SMT} first corrupts every party in C_1 and C_2 except randomly chosen P_i of C_1 and randomly chosen P_j of C_2. In particular, this means that the set of indices J of corrupt parties is some random subset of $[N]$ of size $2n-2$, and index j is some random other index outside of J. Now, the adversary will one by one sample $M = (N-2n+1)/2$ indices $i_1, \ldots, i_M$ from $\mathcal{U}$ that are not already part of J, and add them to J. Let J' be the final such set. From Lemma 3.12, we know that under the above assumption on message size, for any random, fixed-size subset $J \subseteq [N]$ of size $|J| \geq 2n-2$, and random index $j' \notin J$, if we pick another random index $i_l \notin J$, $\mathbb{E}_{J,i_l,j'}[\mathrm{I}(r_{j'}|R_J; r_{i_l})] \geq \lambda/4$. Thus, recalling that $\mathrm{I}(r_{j'}|R_J; r_{i_l}) = \mathrm{H}(r_{j'}|R_j) - \mathrm{H}(r_{j'}|R, r_{i_l})$, we can write

$$\mathbb{E}_{J',j}[\mathrm{I}(r_j; R_{J'})] = \mathbb{E}_{J',j}[\mathrm{H}(r_j) - \mathrm{H}(r_j|R_{J'})] =$$

$$\mathbb{E}_j[\mathrm{H}(r_j)] + \mathbb{E}_{J',j,i_M}[-\mathrm{H}(r_j|R_{J'}) + \mathrm{H}(r_j|R_{J'} \setminus \{r_{i_M}\})]$$

$$+\mathbb{E}_{J',j,i_M,i_{M-1}}[-\mathrm{H}(r_j|R_{J'} \setminus \{r_{i_M}\}) + \mathrm{H}(r_j|R_{J'} \setminus \{r_{i_M}, r_{i_{M-1}}\})]$$

$$\ldots$$

$$+\mathbb{E}_{J',j,i_M,\dots,i_1}[-\mathrm{H}(r_j|R_{J'} \setminus \{r_{i_l}\}_{l\in[2,M]}) + \mathrm{H}(r_j|R_{J'} \setminus \{r_{i_l}\}_{l\in[M]})]$$

$$-\mathbb{E}_{J',j,i_M,\dots,i_1}[\mathrm{H}(r_j|R_{J'} \setminus \{r_{i_l}\}_{l\in[M]}])]$$

$$\geq M \cdot \lambda/4 + \mathbb{E}_{J',j,i_M,\dots,i_1}[\mathrm{H}(r_j) - \mathrm{H}(r_j|R_{J'} \setminus \{r_{i_l}\}_{l\in[M]}) \geq M \cdot \lambda/4.$$

Now, these indices J' will correspond to the parties that $\mathcal{A}$ will corrupt in the execution. However, if some randomly chosen index i_l is indeed the index i corresponding to the only honest party P_i of C_1, the attack will fail, as $\mathcal{A}$ cannot corrupt P_i. Yet, the probability that this happens corresponds to the probability that if we pick M items at random from $N-2n+1$ total items, i is not one of them, which is equal to: $\frac{\binom{N-2n}{M}}{\binom{N-2n+1}{M}} = 1 - \frac{M}{N-2n+1} = 1/2$, since we choose $M = (N-2n+1)/2$.

So, if $\mathbb{E}[|r_j|] \leq (N-2n+1)\cdot\lambda/8$ as in the Theorem statement, in expectation, $\mathcal{A}$ can sample r_j conditioned on $R_{J'}$ correctly (i.e., with probability 1) and guess $c_{i,j}$ with probability greater than $2^{-\lambda/2-1}$. Using the guessed r_j and $c_{i,j}$ along with the learned $\{c_{i',j}\}_{i'\in[n]\setminus\{i\}}$ via corruptions, we can reconstruct B_j. Finally, using $\{B_j\}_{j\in[n+1,2n]}$, we can successfully reconstruct x.

Thus, it cannot be true that $\Pr_{C_1,C_2,i,j}[|c_{i,j}| < \lambda/2] > 1/2$. By the law of total probability, it must therefore be that:

$$\mathbb{E}_{C_1,C_2}[\mathsf{Comm}] = \sum_{i,j} \mathbb{E}_{C_1,C_2}[|c_{i,j}|]$$

$$\geq \sum_{i,j} \lambda/2 \cdot \Pr_{C_1,C_2}[|c_{i,j}| \geq \lambda/2] = \lambda/2 \cdot \sum_{i,j} \Pr_{C_1,C_2}[|c_{i,j}| \geq \lambda/2] \geq \frac{n^2\cdot\lambda}{4}.$$

□

3.4 Honest Majority Protocol

3.4.1 Technical Overview

Here we comment briefly on how we obtain our results in the honest majority setting. We remark that, for the purpose of this overview, we present our results using as a starting point the previous discussion on dishonest majority in Section 3.2.1. In the work of [Choudhuri et al. 2021], honest majority fluid MPC is achieved by letting the parties in a given committee C_i hold *Shamir* sharings of the intermediate circuit values $[x_1]_t^{C_i}, \dots, [x_\ell]_t^{C_i}$ in the i-th layer, where $t < n/2$. To preserve the invariant observe that, because of the multiplicative properties of Shamir secret-sharing, the parties in C_i can *locally* obtain sharings of every intermediate value $[y_1]_{t_1'}^{C_i}, \dots, [y_{\ell'}]_{t_1'}^{C_i}$ in the next layer, where each degree t_j' is either equal to t (for addition and identity gates), or $2t$, which is less than n (for multiplication gates). At this point, the parties in C_i can *reshare* these shared values towards committee C_{i+1}, who obtains $[y_1]_{t_1'}^{C_{i+1}}, \dots, [y_{\ell'}]_{t_{\ell'}'}^{C_{i+1}}$, hence maintaining the invariant.

While in the dishonest majority setting resharing additively shared values (with no authentication) can be achieved with linear communication complexity assuming certain form of committee-agnostic preprocessing, such approach does not work in our current setting. Here, Shamir secret-sharing is used, and resharing in one round requires a quadratic amount of communication as it is done by each party in C_i distributing shares of their Shamir share to each party in C_{i+1}, which can be aggregated to obtain Shamir shares of the underlying secret. This is indeed the approach taken in [Choudhuri et al. 2021], and this is one of the fundamental reasons for the quadratic communication in that work. A second reason is also similar to the one in the dishonest majority setting, and it is related to the reconstruction of secret-shared values.

We can interpret our protocol in the honest majority setting as addressing the two issues highlighted above using some techniques from the dishonest majority case as a base, while adding other new ones, and for the purpose of this section, we describe our protocol in these terms. In a bit more detail, we overcome the issue of resharing with squared communication by, instead of using Shamir secret-sharing with degree $t < n/2$, using a larger degree $n - 1$, which is in essence equivalent to additive secret-sharing, as used in the dishonest majority setting. In principle, this would enable us to perform resharing with linear communication by using preprocessed data as sketched in Section 3.2.1.1. However, an important challenge in the honest majority setting is that we should not use any preprocessing whatsoever since, unlike the dishonest majority setting, it is not required.

Due to the above, our approach for resharing degree-$(n-1)$ Shamir sharings *without preprocessing* with linear communication is different. Assume committee C_i has sharings $[x]^{C_i}_{n-1}$, and the goal is for committee C_{i+1} to obtain $[x]^{C_{i+1}}_{n-1}$. Let us write $C_i = \{P_1, \dots, P_n\}$ and $C_{i+1} = \{Q_1, \dots, Q_n\}$, and also $[x]^{C_i}_{n-1} = (x^1, \dots, x^n)$. Assume the parties in C_i have preprocessed a sharing of zero $[0]^{C_i}_{n-1} = (o^1, \dots, o^n)$.[8] Our resharing protocol is summarized as follows: each party P_j sends $x^j + o^j$ to Q_j, and committee C_{i+1} defines $[x]^{C_{i+1}}_{n-1}$ to be these received shares. In words, shares are transferred in a "straight line fashion" (after randomizing with shares of zero), and the new sharings are exactly *the same* as the previous ones. This approach does not work in the dishonest majority setting: the adversary can corrupt, say, $P_1, \dots, P_{n-1}$ in the first committee, and by corrupting Q_n the adversary learns all shares. In contrast, in the honest majority setting, the adversary learns at most t shares in the first committee and t shares in the second, for a total of $\leq 2t < n$ shares, which maintain privacy of the underlying secret. This powerful observation turns out to be the enabling tool for linear communication.

Using degree-$(n-1)$ Shamir sharings means that the shares of the honest parties in a given committee do not determine the underlying secret anymore, which enables a corrupt party to cheat by modifying their share. Importantly, a similar issue was faced in the dishonest majority setting with additive secret-sharing, and fortunately we are able to take a similar approach here by using MACs in order to prevent cheating. We remark that these are not needed in [Choudhuri et al. 2021], since they use Shamir sharings of low degree. We do not elaborate on how MACs are used in our protocol to prevent cheating, but we mention that the approach is in spirit similar to the one sketched in the dishonest majority overview.

The final details we comment on are related to the "preprocessing" required in our protocol. As we mentioned initially, it is imperative that our honest majority protocol does not make use

[8]As we elaborate on below, this type of preprocessing can in fact be generated "on the fly" by the different committees, so it is not considered preprocessing as such.

of any preprocessing material. However, we already mentioned some form of preprocessing (namely, shares of zero $[0]_{n-1}^{C_i}$), plus, several ideas from the dishonest majority protocol require preprocessing such as authenticated values $([r]_{n-1}^{C_i}, \left[\Delta_{C_i} \cdot r\right]_{n-1}^{C_i}, \left[\Delta_{C_i}\right]_{n-1}^{C_i})$, or authenticated multiplication triples. Fortunately, in our work we are able to leverage once more the fact that we have an honest majority in order to let committee C_{i-1} generate the "preprocessing" for committee C_i *on the fly*. For correlations that are "linear" such as sharings of zero, the approach from [Damgård and Nielsen 2007] can be easily adapted, where the parties in committee C_{i-1} distribute sharings to C_i, and the latter perform randomness extraction using a Vandermonde matrix. On the other hand, for correlations that include a multiplication, like multiplication triples or authenticated values $([r]_{n-1}^{C_i}, \left[\Delta_{C_i} \cdot r\right]_{n-1}^{C_i}, \left[\Delta_{C_i}\right]_{n-1}^{C_i})$, the parties in C_{i-1} can obtain the linear part $([r]_t^{C_{i-1}}, \left[\Delta_{C_i}\right]_t^{C_{i-1}})$ from C_{i-2} using the ideas we just described for linear correlations (notice the degree is $t < n/2$). Then, the parties in C_{i-1} locally multiply these sharings, to obtain $\left[\Delta_{C_i} \cdot r\right]_{2t}^{C_{i-1}}$. Finally, the parties in C_{i-1} perform the "straight-line" resharing from before so that C_i obtains $([r]_{n-1}^{C_i}, \left[\Delta_{C_i} \cdot r\right]_{n-1}^{C_i}, \left[\Delta_{C_i}\right]_{n-1}^{C_i})$.

3.4.2 Formal Protocol

We now turn to presenting our protocol for fluid MPC with linear communication complexity and maximal fluidity in the honest majority setting, where each committee contains at most a minority of corrupt parties. The outline of this section is the following. First, in Section 3.4.2.2, we present a major building block, Procedure $\pi_{\text{eff-reshare-hm}}$, which enables a given committee holding a sharing of a random value to efficiently reshare this secret to the next committee. As in the two previous fluid protocols [Rachuri and Scholl 2022; Choudhuri et al. 2021], we make use of a randomized version of the circuit that aims at detecting cheating in multiplication gates, and we also draw inspiration from [Rachuri and Scholl 2022] and make use of a MAC check that accounts for the correctness of the openings throughout the computation, which is crucial in our case to achieve linear communication complexity. This is discussed in Section 3.4.2.3. Then, in Section 3.4.2.4

we show how the parties make progress through the computation by processing multiplication gates. Finally, these pieces are put together in Section 3.4.2.5 to obtain our final protocol, $\Pi_{\text{main-hm}}$, for honest majority MPC in the fluid model with linear communication complexity and maximal fluidity.

3.4.2.1 Initial building blocks.

We now present some of the building blocks we will require for our final protocol. For our main honest majority protocol, we will require the following functionalities. These are fairly standard in the literature and implementing them in the fluid setting represents little challenge, using the randomness extraction ideas through Vandermonde matrices in [Damgård and Nielsen 2007]. Thus we omit their instantiations for brevity.

Figure 3.13: Functionality $\mathcal{F}_{\text{rand-hm}}$

Functionality: Distribute degree-t_i sharings of random value r to C_i.

1. $\mathcal{F}_{\text{rand-hm}}$ receives from the adversary shares $\{r_i\}_{i\in\mathcal{T}_C}$. $\mathcal{F}_{\text{rand-hm}}$ views these as the shares of the corrupted parties.
2. $\mathcal{F}_{\text{rand-hm}}$ randomly samples r, then based on r and the t_i shares $\{r_i\}_{i\in\mathcal{T}_C}$ of corrupted parties, $\mathcal{F}_{\text{rand-hm}}$ reconstructs the whole sharing $[r]_{t_i}^{C_i}$.
3. Finally, $\mathcal{F}_{\text{rand-hm}}$ distributes the shares of $[r]_{t_i}^{C_i}$ to the honest parties of C_i.

Figure 3.14: Functionality $\mathcal{F}_{\text{coin}}$

Functionality: Sample a random coin $r \leftarrow_{\$} \mathbb{F}_p$ to C_i.

1. $\mathcal{F}_{\text{coin}}$ samples a random field element r.
2. $\mathcal{F}_{\text{coin}}$ sends r to the adversary and:
    - If the adversary replies continue, $\mathcal{F}_{\text{coin}}$ sends r to the honest parties of C_i.
    - If the adversary replies abort, $\mathcal{F}_{\text{coin}}$ sends abort to the honest parties of C_i.

Figure 3.15: Functionality $\mathcal{F}_{\text{double-rand-hm}}$

Functionality: Distribute degree-t_i and degree-$2t_i$ sharings of the same random value r to C_i.

1. $\mathcal{F}_{\text{double-rand-hm}}$ receives from the adversary two sets of shares $\{r_i\}_{i\in\mathcal{T}_C}$ and $\{r'_i\}_{i\in\mathcal{T}_C}$. $\mathcal{F}_{\text{double-rand-hm}}$ views the first set as the shares of the corrupted parties for the degree t_i-sharing, and the second set as the shares for the degree $2t_i$-sharing.

2. $\mathcal{F}_{\text{double-rand-hm}}$ randomly samples r and prepares the double sharings as follows.

   - For the degree-t_i sharing, based on the secret r and the t_i shares $\{r_i\}_{i\in\mathcal{T}_C}$ of corrupted parties, $\mathcal{F}_{\text{double-rand-hm}}$ reconstructs the whole sharing $[r]_{t_i}^{C_i}$.

   - For the degree-$2t_i$ sharing, $\mathcal{F}_{\text{double-rand-hm}}$ randomly samples t_i elements as the shares of the first t_i honest parties. Based on the secret r, the t_i shares of the first t_i honest parties, and the t_i shares $\{r'_i\}_{i\in\mathcal{T}_C}$ of the corrupted parties, $\mathcal{F}_{\text{double-rand-hm}}$ reconstructs the whole sharing $[r]_{2t_i}^{C_i}$.

3. Finally, $\mathcal{F}_{\text{double-rand-hm}}$ distributes the shares of $([r]_{t_i}^{C_i}, [r]_{2t_i}^{C_i})$ to the honest parties of C_i.

Figure 3.16: Functionality $\mathcal{F}_{\text{zero}}$

Functionality: Distribute degree $2t_i$ shares of $o = 0$ to C_i.

1. $\mathcal{F}_{\text{zero}}$ receives from the adversary the set of shares $\{r_i\}_{i\in\mathcal{T}_C}$.

2. $\mathcal{F}_{\text{zero}}$ randomly samples t_i elements as the shares of the first t_i honest parties. Based on the secret $o = 0$, the t_i shares of the first t_i honest parties, and the t_i shares $\{r_i\}_{i\in\mathcal{T}_C}$ of the corrupted parties, $\mathcal{F}_{\text{zero}}$ reconstructs the whole sharing $[o]_{2t_i}^{C_i}$.

3. Finally, $\mathcal{F}_{\text{zero}}$ distributes the shares of $[o]_{2t_i}^{C_i}$ to the honest parties of C_i.

As we will later accomplish in $\Pi_{\text{main-hm}}$, each committee will have a degree-t_i and degree-$2t_i$ double sharing of the global MAC key Δ. Therefore we will assume that all procedures presented below that are invoked by C_i will implicitly take these sharings as input.

We rely on the following procedure that enables the parties in a given committee C_i to obtain authenticated sharings of a uniformly random value $[\![r]\!]^{C_i}$, assuming Shamir sharings of the key $([\Delta]_{t_i}^{C_i}, [\Delta]_{2t_i}^{C_i})$. This is described below. Observe that in the protocol the MAC sharings produced $[r \cdot \Delta]_{t_i}^{C}$ are not uniformly random, but instead, they are a product $[r]_{t_i}^{C_i} \cdot [\Delta]_{t_i}^{C_i}$. These sharings will be randomized in the places we use them.

Figure 3.17: Procedure $\pi_{\text{get-rand-sharing}}$

Usage: Using double sharing $([\Delta]_{t_i}^{C_i}, [\Delta]_{2t_i}^{C_i})$ of the global MAC key, C_i outputs a random SPDZ sharing $[\![r]\!]^{C_i}$.

1. All parties in C_i invoke $\mathcal{F}_{\text{double-rand-hm}}$ to get random double sharing $([r]_{t_i}^{C_i}, [r]_{2t_i}^{C_i})$.
2. Parties in C_i then locally obtain and output authenticated sharing $[\![r]\!]^{C_i} \leftarrow ([r]_{2t_i}^{C_i}, [r]_{t_i}^{C_i} \cdot [\Delta]_{t_i}^{C_i}, [\Delta]_{2t_i}^{C_i})$.

3.4.2.2 Efficient Resharing for Honest Majority

As we highlighted in Section 3.4.1, a fundamental reason why the protocol from [Choudhuri et al. 2021] does not achieve linear communication complexity stems from the fact that the hand-off procedure from one committee to the next one consists of every party resharing their share towards the next committee, which requires quadratic communication. In our work, we address this limitation by making use of Procedure $\pi_{\text{eff-reshare-hm}}$ below, which shows how to reshare a degree-$2t_i$ Shamir sharing from committee C_i to the next committee C_{i+1}, while using only linear communication. The idea is in fact simple: assuming each committee has the same amount of parties n (the procedure below is more general), each party with index j in committee C_i will send (a re-randomized version of) their share to the party with index j in C_{i+1} directly. This is secure since the adversary learns in total at most $2t$ shares across the two committees, which is the degree of the polynomial used. As briefly mentioned above, the parties first re-randomize their shares using $\mathcal{F}_{\text{zero}}$, which is done to prevent a new sharing from leaking the underlying secret when transmittted to the next committee.

Figure 3.18: Procedure $\pi_{\text{eff-reshare-hm}}$

Usage: C_i reshares re-randomized $[r]^{C_i}_{2t_i}$ to C_{i+1}. Assume that the parties in C_i are indexed from 1 to n_i and those in C_{i+1} are indexed from $n_i + 1$ to $n_i + n_{i+1}$.

1. Let $[r]^{C_i}_{2t_i}$ be the input shares.
2. C_i invokes $\mathcal{F}_{\text{zero}}$ and receives a sharing of $o = 0$, $[o]^{C_i}_{2t_i}$.
3. All parties locally compute $[r']^{C_i}_{2t_i} \leftarrow [r]^i_{2t_i} + [o]^i_{2t_i}$, for $r' = r + 0 = r$.
4. Finally:
   - If $n_i < n_{i+1}$: Let $d = n_{i+1}/n_i$ (assuming $n_i | n_{i+1}$ for simplicity). Each $P_j \in C_i$ samples $d-1$ random values r_l, sets $r_{j\cdot d} \leftarrow (r')^j - \sum_{l=1}^{d-1} r_l$, where $(r')^j$ is their share of $[r']^{C_i}_{2t_i}$, and sends each r_l for $l \in [d]$ to $P_{n_i+(j-1)\cdot d+l} \in C_{i+1}$, who outputs this as their share of $[r']^{C_{i+1}}_{2t_{i+1}}$.
   - Else: Let $d = n_i/n_{i+1}$ (assuming $n_{i+1}|n_i$ for simplicity). For l such that $(l-1)\cdot d < j \leq l \cdot d$, each $P_j \in C_i$ sends their share r'^j to $P_{n_i+l} \in C_{i+1}$, who outputs as their share of $[r']^{C_{i+1}}_{2t_{i+1}}$, $\sum_j r'^j$ for each P_j it received from.

Lemma 3.13. *Assume that at most $2t_i$ shares of $[r]^{C_i}_{2t_i}$ can be computed by $\mathcal{A}$ (and the rest are uniformly random to $\mathcal{A}$). Then procedure $\pi_{\text{eff-reshare-hm}}$'s transcript is simulatable with random values and preserves the invariant that at most $2t_{i+1}$ shares of $[r]^{C_{i+1}}_{2t_{i+1}}$ can be computed by $\mathcal{A}$, while the rest are uniformly random to $\mathcal{A}$.*

Proof. For this proof, we assume for simplicity that $t_i = t_{i+1}$. Now, assume w.l.o.g. that the shares of $[r]^{C_i}_{2t_i}$ known by $\mathcal{A}$ are $r^1, \ldots, r^{2t_i}$, which must also mean that $P_{n_i} \in C_i$ is honest. Thus, we can also assume w.l.o.g. that the corrupted parties in C_i are $P_1, \ldots, P_{t_i}$. This means that the shares of $[o]^{C_i}_{2t_i}$ held by $P_{t_i+1}, \ldots, P_{n_i}$, are unknown and uniformly random (subject to them reconstructing to 0) to $\mathcal{A}$, by the security of $\mathcal{F}_{\text{zero}}$.

Now, consider what the adversarial parties in $\mathcal{T}_{C_{i+1}}$ are sent from $P_{t_i+1} \dots P_{n_i}$: $r^j + o^j$ (where each $P_j \notin \mathcal{T}_{C_i}$ in the worst case). Since $\mathcal{A}$ does not know at least two shares of $[o]_{2t_i}^{C_i}$, the o^j's in the communication above that it receives are each individually uniformly random. Therefore, all of these messages can be simulated with random values.

In particular, this means that even if the adversary sees $r^{n_i} + o^{n_i}$, r^{n_i} remains uniformly random and unknown to $\mathcal{A}$. Furthermore, consider some $P_l \in C_{i+1}$ such that P_l itself is uncorrupted and $P_j \in C_i$ who sent to P_l was not corrupted (there must exist at least one such pair). Since the o^j in P_l's share $(r')^l = r^j + o^j$ is uniformly random and unknown to $\mathcal{A}$, then $(r')^l$ itself is uniformly random and unknown to $\mathcal{A}$, even if r^j was known by $\mathcal{A}$. In fact, all of the shares in this case are uniform and unknown to $\mathcal{A}$. All other shares (corresponding to the case in which at least one of $P_l \in C_{i+1}$, or the P_j who sent to P_l is corrupted) are known to $\mathcal{A}$. Thus, the invariant is preserved. □

Inefficient resharing. We will also need to reshare degree-t_i Shamir sharings across committees using Procedure $\pi_{\text{ineff-reshare-hm}}$, below. This can only be done with $\Omega(n^2)$ communication, however, since it is only done once per committee, we can still achieve $O(n|C|)$ total communication if the width of circuit C is $\Omega(n)$.

Figure 3.19: Procedure $\pi_{\text{ineff-reshare-hm}}$

Usage C_i reshares $[r]_{t_i}^{C_i}$ to C_{i+1}.

1. Let r^j be P_j's share of $[r]_{t_i}^{C_i}$. Each $P_j \in C_i$ will create a random degree t_{i+1} Shamir secret sharing $\left[r^j\right]_{t_{i+1}}^{C_i}$ of their share and distribute the corresponding shares to each $P_l \in C_{i+1}$.
2. Finally, each $P_l \in C_{i+1}$ will then compute $[r]_{t_{i+1}}^{C_i} \leftarrow \sum_{j \in C_i} c_j \left[r^j\right]_{t_{i+1}}^{C_i}$, where c_j is the Lagrange reconstruction coefficient for a degree-t_{i+1} polynomial.

Lemma 3.14. *Procedure $\pi_{\text{ineff-reshare-hm}}$'s transcript is simulatable with random values.*

Proof. This follows easily from the fact that the shares of the honest parties of C_i are unknown to the adversary. So, by the security of Shamir secret sharing, the t_{i+1} shares of each honest party's share in C_i that the corrupt parties of C_{i+1} receive are uniformly random. □

3.4.2.3 Incremental Checks

As in [Choudhuri et al. 2021], we achieve active security by maintaining a few "accumulators" that somehow aggregate the potential errors that are introduced by each committee. These accumulators are updated by every other committee, and the current (possibly updated) version of the accumulator is transferred from one committee to the next. Finally, the final committees will use these accumulators to verify the integrity of the computation.

In our protocol, we make use of the "straightline" resharing procedure $\pi_{\text{eff-reshare-hm}}$ that achieves linear communication complexity, but requires a larger threshold of $2t_i$ to achieve security. This means that the underlying secrets are not determined by the honest parties alone, and as a result a malicious adversary can in fact add errors to *any* value throughout the computation. A similar issue happens in the dishonest majority fluid protocol of [Rachuri and Scholl 2022], and we draw inspiration from such approach to address this attack in our protocol. The solution consists of using MACs, which are used to authenticate every intermediate value used throughout the computation and serve as additional redundancy on secret values that guarantees integrity. This is done by maintaining an accumulator that attests for the integrity of all of the reconstructions, which is built using the shared MACs and the claimed openings.

Succinctly maintaining this accumulator involves opening random challenges β to committees, who then use such β to compute random linear combinations that *compress* the verification of many MACs into one field element that should be 0. However, these challenges β should not be opened at the same time that the values whose MACs it checks are opened, for otherwise the adversary could cheat in the above linear combination. Thus, when the parties of C_i receive some openings and want to verify their MACs, they each hash together all of these openings and then send these hashes to each of the parties of C_{i+1}. The challenge β is then opened to C_{i+1}, and only P_1 of C_i forwards all of the openings to all of the parties of C_{i+1}, in order to maintain linear communication. Since the hashes prevent P_1 from changing the openings, they cannot be dependent on β. Since the hashes are short, total communication will still be $O(n|C|)$ if the width of C is $\Omega(n)$.

Also note that it takes two committees to update the accumulator based on values opened to the first committee. However, since values are only opened to every other committee, there is no entanglement of updates. Details are given in Procedure $\pi_{\text{MAC-check-hm}}$ below.

Figure 3.20: Procedure $\pi_{\text{MAC-check-hm}}$

Usage: Each committee C_i incrementally updates a MAC check state $[\sigma]_{2t_i}^{C_i}$ based on the values opened to them, which the final committees at the end of the computation use to check that all openings throughout the protocol were performed correctly.

Init: Each Party P_i in committee C_4 (the first to have values opened to it, since the first invocation of $\pi_{\text{mult-hm}}$ is by C_2 to create randomized versions of the circuit inputs) initially defines their share of $[\sigma]_{2t_4}^{C_4}$ as $\sigma^i \leftarrow 0$.

Update State: On input (update, $\{(A_m, [\Delta \cdot A_m]_{2t_i}^{C_i})\}_{m\in[T]}, [\Delta]_{2t_i}^{C_i}$) from committee C_i, where $\{A_m\}_{m\in[T]}$ were the values opened to C_i:

1. First each party $P_j \in C_i$ samples keys $s_{j,l}$ to the universal hash family $\mathsf{H} = \{\mathsf{H}_s : \mathbb{F}_p^T \to \mathbb{F}_p\}$ for each $P_l \in C_{i+1}$ and computes $h_{j,l} = \mathsf{H}_{s_{j,l}}(\{A_m\}_{m\in[T]})$.

2. In parallel: (i) each party $P_j \in C_i$ then sends to each $P_l \in C_{i+1}$ the universal hash key and value $s_{j,l}, h_{j,l}$; (ii) only P_1 sends $\{A_m\}_{m\in[T]}$ to each $P_l \in C_{i+1}$; and (iii) all of C_i invokes $\pi_{\text{eff-reshare-hm}}$ on $[\sigma]_{2t_i}^{C_i}, \{[\Delta \cdot A_m]_{2t_i}^{C_i}\}_{m\in[T]}$.

3. Each P_l in Committee C_{i+1} first for each $P_j \in C_i$ computes $h'_{j,l} \leftarrow \mathsf{H}_{s_{j,l}}(\{A_m\}_{m\in[T]})$ and checks if $h'_{j,l} = h_{j,l}$. If not, it aborts; else continues.

4. C_{i+1} then invokes $\mathcal{F}_{\text{coin}}$ to get a random challenge β.

5. Each $P_l \in C_{i+1}$ next locally computes $A \leftarrow \sum_{m=1}^{T} \beta^m \cdot A_m$ and $[\gamma]_{2t_{i+1}}^{C_{i+1}} \leftarrow \sum_{m=1}^{T} \beta^m \cdot [\Delta \cdot A_m]_{2t_{i+1}}^{C_i}$.

6. It finally updates $[\sigma]_{2t_{i+1}}^{C_{i+1}} \leftarrow [\sigma]_{2t_{i+1}}^{C_{i+1}} + [\gamma]_{2t_{i+1}}^{C_{i+1}} - [\Delta]_{2t_{i+1}}^{C_{i+1}} \cdot A$ and invokes $\pi_{\text{eff-reshare-hm}}$ on $[\sigma]_{2t_{i+1}}^{C_{i+1}}$.

Check State: On input check from committee C_i:

1. Let σ^j be the share of the MAC check state $[\sigma]_{2t_i}^{C_i}$ held by each $P_j \in C_i$. Each P_j creates a random degree-t_{i+1} Shamir secret share $\left[\sigma^j\right]_{t_{i+1}}^{C_{i+1}}$ of their share σ^j and distributes the corresponding shares to the parties of C_{i+1}.

2. Then each party $P_l \in C_{i+1}$ computes $[\sigma]_{t_{i+1}}^{C_{i+1}} \leftarrow \sum_{j \in C_i} c_j \cdot \left[\sigma^j\right]_{t_{i+1}}^{C_{i+1}}$, where c_j is the Lagrange reconstruction coefficient for a degree-$2t_i$ polynomial.

3. Finally, parties open the shares of $[\sigma]_{t_{i+1}}^{C_{i+1}}$ to each party of C_{i+2}, who reconstruct σ, and if successful, output Accept if $\sigma = 0$; else Reject.

Lemma 3.15. *Procedure $\pi_{\text{MAC-check-hm}}$ is correct, i.e., it accepts if all the opened values A_m and the corresponding MACs are computed correctly. Moreover, it is sound, i.e., it rejects except with probability at most $(2 + \max_i T_i)/p$ in case at least one opened value is not correctly computed. Furthermore, the transcript of* ***Update State*** *is simulatable.*

Proof. The proof follows along the lines of the proof for Lemma 3.4 in the dishonest majority case. One difference is that the parties in C_i send unique universal hash keys to each party in C_{i+1}, rather than getting them from the preprocessing of the dishonest majority protocol. But since there will be at least one honest party in each committee, at least one key will remain random and unknown to the adversary, and thus it serves the same purpose. Also, $\pi_{\text{MAC-check-hm}}$ gets β from $\mathcal{F}_{\text{coin}}$, but from the security of $\mathcal{F}_{\text{coin}}$, this is also still random and independent of all other values.

So, the adversary can thus only inject additive error δ_m^i for each m-th value A_m^i opened to C_i, η^i for when the MAC key Δ is reshared to C_i, ε_m^i for when the MAC of the m-th opened value is reshared to C_i, and ζ^i for when the current accumulator value σ is reshared to C_i. Additionally, in the **Check State** phase, since at least one share of $[\sigma]_{2t_i}^{C_i}$ is unknown to $\mathcal{A}$ and remains unknown after the degree reduction step, $\mathcal{A}$ can only inject another additive error ε independent of σ. Thus, ensuring that $[\sigma]_{2t_\ell}^{C_\ell} = 0$ follows the analysis of the proof of Lemma 3.4 for the dishonest majority case.

Finally, we know from Lemma 3.13 that $\pi_{\text{eff-reshare-hm}}$ is simulatable by random values. Also, the universal hash keys are sampled randomly by the clients, and the opened values are known to the simulator, so the hash keys and their resulting hash outputs can easily be simulated. □

Unfortunately, this is not the only kind of error we need to account for. As in [Rachuri and Scholl 2022], the c parts of multiplication triples that are used in $\pi_{\text{mult-hm}}$ are only authenticated "on the fly". This means that the adversary can inject additive errors into these c parts that $\pi_{\text{MAC-check-hm}}$ will not catch (since the corresponding errors will be incorporated into the MACs, too). As a result, multiplications may not be computed correctly. To address this attack vector, we use similar ideas to [Rachuri and Scholl 2022; Choudhuri et al. 2021], which have their roots in the techniques of [Chida et al. 2018], and consists of maintaining a randomized version of the circuit which can be used to verify multiplications. The associated accumulator is presented in Procedure $\pi_{\text{mult-verify-hm}}$ below.[9]

Figure 3.21: Procedure $\pi_{\text{mult-verify-hm}}$

Usage: Each committee C_i that gets the output wires of the multiplication gates of some layer ℓ of the circuit incrementally updates a multiplication verification state $(\llbracket u' \rrbracket^{C_{2t_i}}, \llbracket w' \rrbracket^{C_{2t_i}})$, which the final committees at the end of the computation use to check that all multiplications throughout the protocol were performed correctly.

Init: Each Party P_i in committee C_4 (the first to get output from $\pi_{\text{mult-hm}}$, as a result of C_2 creating randomized versions of the circuit inputs) initially defines their shares of $\llbracket u' \rrbracket^{C_4}, \llbracket w' \rrbracket^{C_4}$ as $(u')^i =$

[9]Note that the invocations of $\pi_{\text{MAC-check-hm}}$ in the **Check State** phase of $\pi_{\text{mult-verify-hm}}$ can be condensed to 3 rounds, since only one value at a time is opened.

$(w')^i \leftarrow 0$ (same for the MAC shares).

Update State: On input $(\text{update}, \{(\llbracket z_m \rrbracket^{C_i}, \llbracket rz_m \rrbracket^{C_i}\}_{m\in[T]})$ from committee C_i, where $\{(\llbracket z_m \rrbracket^{C_i}, \llbracket rz_m \rrbracket^{C_i}\}_{m\in[T]}$ were the output wires of multiplication gates computed by C_i:

1. Each P_j in C_i invokes $\mathcal{F}_{\text{coin}}$ to get random challenge α.
2. Parties P_j in C_i locally compute $\llbracket u \rrbracket^{C_i} \leftarrow \llbracket u \rrbracket^{C_i} + \sum_{m=1}^{T} \alpha^m \cdot \llbracket rz_m \rrbracket^{C_i}$ and $\llbracket w \rrbracket^{C_i} \leftarrow \llbracket w \rrbracket^{C_i} + \sum_{m=1}^{T} \alpha^m \cdot \llbracket z_m \rrbracket^{C_i}$.
3. Finally C_i invokes $\pi_{\text{eff-reshare-hm}}$ on $\llbracket u \rrbracket^{C_i}, \llbracket w \rrbracket^{C_i}$.

Check State: On input check from the clients C_i:

1. The parties of C_i first open $\llbracket r \rrbracket^{C_i}$ to the parties of C_{i+1}, who then check its MAC by running both phases of $\pi_{\text{MAC-check-hm}}$ on it.
2. Then the parties of C_{i+4} ($\pi_{\text{MAC-check-hm}}$ takes 4 rounds) all open $(\llbracket u \rrbracket^{C_{\text{clnt}}} - r \cdot \llbracket w \rrbracket^{C_{\text{clnt}}})$ to the parties of C_{i+5}, who then check its MAC by running both phases of $\pi_{\text{MAC-check-hm}}$ on it. If the opened value is 0, the parties of C_{i+8} ($\pi_{\text{MAC-check-hm}}$ takes 4 rounds) output Accept; else Reject.

Lemma 3.16. *Procedure $\pi_{\text{mult-verify-hm}}$ is correct, i.e., it accepts if all multiplications are computed correctly. Moreover, it is sound, i.e., it rejects except with probability at most $(1 + \max_i T_i)/p$ in case at least one multiplication is not correctly computed. Furthermore, the transcript of* ***Update State*** *is simulatable.*

Proof. This proof follows along the lines of Lemma 3.8 for the dishonest majority case. One difference is that in $\pi_{\text{mult-hm}}$, we do not need to use a universal hash function. This is because we can authenticate $\llbracket c \rrbracket^{C_{i-2}}$ before $(x + a), (y + b)$ are opened and thus the error in $\llbracket c \rrbracket^{C_{i-2}}$ must be independent of them (and also the later opened challenge α). So, the probability of $\pi_{\text{mult-verify-hm}}$ failing when the adversary cheats is even lower (i.e., as in the lemma statement). Also, since parties locally compute $[c]^{C_i}_{2t_i} \leftarrow [a]^{C_i}_{t_i} \cdot [b]^{C_i}_{t_i}$, there is only additive error on c, δ_c, i.e., independent of the shares a^j, b^j of the honest parties (and same for $[\![c']\!]^{C_{i+2}}$ of the randomized computation). Finally, $\pi_{\text{mult-verify-hm}}$ gets α from $\mathcal{F}_{\text{coin}}$, but from the security of $\mathcal{F}_{\text{coin}}$, α is in this case also random and independent of all other values.

So (assuming that $\pi_{\text{MAC-check-hm}}$ does not fail), the adversary can only inject additive error $\epsilon_{i,m}(= \delta_{c_{i,m}})$ for each m-th multiplication gate that C_i receives output for, along with $\epsilon'_{i,m}$ for that of the randomized version of the multiplication gate, and finally, any accumulated error $\eta_{i,m}$ on the randomized value from the previous gates in the circuit. Thus, ensuring that $[\![u]\!]^{C_{\text{clnts}}} - r \cdot [\![w]\!]^{C_{\text{clnts}}} = 0$ follows the exact same case analysis of that in the proof of Lemma 3.8.

Finally, we know from Lemma 3.13 that $\pi_{\text{eff-reshare-hm}}$ is simulatable by random values. □

3.4.2.4 Secure Multiplication

Finally, before we discuss our ultimate protocol, we present Procedure $\pi_{\text{mult-hm}}$ below which enables a given committee to make progress on the computation by securely processing multiplication gates. At a high level, this procedure makes use of multiplication triples [Beaver 1992] to reduce the task of securely multiplying two shared values, to that of reconstructing two secrets. Reconstruction is done by using the "king idea", originating from [Damgård and Nielsen 2007], which achieves linear communication complexity by first reconstructing to a single party who then sends the reconstruction to the other parties.

However, there are a couple of issues we need to deal with. First, using multiplication triples requires different committees to have access to the same multiplication triple. We indeed achieve this by making use of our resharing procedure $\pi_{\text{eff-reshare-hm}}$ from Section 3.4.2.2. Second, a given committee can only obtain a *partially* authenticated multiplication triple $([\![a]\!]^C, [\![b]\!]^C, [c]^C_{2t})$, where the c part is not authenticated. Using an idea from [Rachuri and Scholl 2022], we authenticate the c part of each triple "on the fly". Intuitively, this is done by masking $[c]^C_{2t}$ with a random, authenticated sharing $[\![v]\!]^C$, reconstructing $(c + v)$, then creating unmasked, authenticated shares of c using $[\![v]\!]^C$ (including its MAC). Reconstructing $(c+v)$ here is also done by using the "king idea". The details are presented below.

Figure 3.22: Procedure $\pi_{\text{mult-hm}}$

Usage: Using double sharing $([\Delta]_{t_i}^{C_i}, [\Delta]_{2t_i}^{C_i})$ of the global MAC key, multiply $[\![x]\!]^{C_i}$ and $[\![y]\!]^{C_i}$ held by C_i so that C_{i+2} outputs $[\![x \cdot y]\!]^{C_{i+2}}$.

1. All parties in C_{i-2} agree on a special party P_{king} in C_{i-1}.
2. All parties in C_{i-2} invoke $\pi_{\text{get-rand-sharing}}$ three times to get $[\![a]\!]^{C_{i-2}}, [\![b]\!]^{C_{i-2}}, [\![v]\!]^{C_{i-2}}$ (they also save the sharings $[a]_{t_{i-2}}^{C_{i-2}}, [b]_{t_{i-2}}^{C_{i-2}}$ generated during this invocation).
3. C_{i-2} then locally obtains (unauthenticated) $[c]_{2t_{i-2}}^{C_{i-2}} \leftarrow [a]_{t_{i-2}}^{C_{i-2}} \cdot [b]_{t_{i-2}}^{C_{i-2}}$.
4. Finally, parties in C_{i-2} in parallel invoke $\pi_{\text{eff-reshare-hm}}$ on input $[\![a]\!]^{C_{i-2}}, [\![b]\!]^{C_{i-2}}, [\![v]\!]^{C_{i-2}}$ and open $[c+v]_{2t_{i-2}}^{C_{i-2}}$ to P_{king} in C_{i-1}.
5. Then, while P_{king} distributes opened $(c+v)$ to the parties of C_i, the rest of the parties in C_{i-1} in parallel invoke $\pi_{\text{eff-reshare-hm}}$ on input $[\![a]\!]^{C_{i-1}}, [\![b]\!]^{C_{i-1}}, [\![v]\!]^{C_{i-1}}$.
6. Parties in C_i then use opened $(c+v)$ to compute authenticated $[\![c]\!]^{C_i} \leftarrow ((c+v) - [v]_{2t_i}^{C_i}, [\Delta]_{2t_i}^{C_i} \cdot (c+v) - [\Delta \cdot v]_{2t_i}^{C_{i+2}}, [\Delta]_{2t_i}^{C_i})$.
7. Parties in C_i agree on a special party P'_{king} in C_{i+1} and then compute $[\![x+a]\!]^{C_i} \leftarrow [\![x]\!]^{C_i} + [\![a]\!]^{C_i}$, and $[\![y]\!]^{C_i} \leftarrow [\![y]\!]^{C_i} + [\![b]\!]^{C_i}$.
8. Parties in C_i then in parallel open $[\![x+a]\!]^{C_i}$, $[\![y+b]\!]^{C_i}$ to P'_{king} in C_{i+1} and invoke $\pi_{\text{eff-reshare-hm}}$ on $[\![a]\!]^{C_i}, [\![b]\!]^{C_i}, [\![c]\!]^{C_i}, [\Delta \cdot (x+a)]_{2t_i}^{C_i}, [\Delta \cdot (y+b)]_{2t_i}^{C_i}$.
9. Then while P'_{king} distributes opened $d = x+a$ and $e = y+b$ to the parties of C_{i+2}, all parties in C_{i+1} invoke $\pi_{\text{eff-reshare-hm}}$ on input $[\![a]\!]^{C_{i+1}}, [\![b]\!]^{C_{i+1}}, [\![c]\!]^{C_{i+1}}, [\Delta \cdot (x+a)]_{2t_{i+1}}^{C_{i+1}}, [\Delta \cdot (y+b)]_{2t_{i+1}}^{C_{i+1}}$.
10. C_{i+2} finally locally computes $[\![x \cdot y]\!]^{C_{i+2}} \leftarrow de - d\,[\![b]\!]^{C_{i+2}} - e\,[\![a]\!]^{C_{i+2}} + [\![c]\!]^{C_{i+2}}$.

11. Parties in C_{i+2} will also invoke $\pi_{\text{MAC-check-hm}}$ on input $(\text{update}, \{((x_m + a_m, [\Delta \cdot (x_m + a_m)]_{2t_{i+2}}^{C_{i+2}}), (y_m + b_m, [\Delta \cdot (y_m + b_m)]_{2t_{i+2}}^{C_{i+2}}))\}_{m \in [T]})$ corresponding to all of the openings of the above form they receive for the multiplication gates at this layer of the circuit.

Lemma 3.17. *Procedure $\pi_{\text{mult-hm}}$'s transcript is simulatable.*

Proof. First, we know that $\pi_{\text{eff-reshare-hm}}$ is simulatable by random values from Lemma 3.13. Also, since a, b, v are uniformly random and unknown to the adversary by the security of $\mathcal{F}_{\text{double-rand-hm}}$, openings $(c + v), (x + a), (y + b)$ are simulatable by random values. Finally, from Lemma 3.15, we know that $\pi_{\text{MAC-check-hm}}$'s transcript is indeed simulatable. □

3.4.2.5 Honest Majority Main Protocol

With all the previous tools into place, we are finally ready to present our full-fledged actively secure, honest majority MPC protocol in the fluid setting, achieving linear communication complexity and maximal fluidity. The clients first distribute double sharings $([x_i]_{t_1}^{C_1}, [x_i]_{2t_1}^{C_1})$ of their inputs to C_1. Then C_1 obtains a double sharing $([\Delta]_{t_1}^{C_1}, [\Delta]_{2t_1}^{C_1})$ of the global MAC key using $\mathcal{F}_{\text{double-rand-hm}}$, and forms authenticated SPDZ sharings of the inputs using these sharings. The committees then proceed to compute both the regular and randomized version of the circuit on the authenticated inputs, using $\pi_{\text{mult-hm}}$ as well as addition and identity gate procedures that work similarly using the same "mask, open to king, and unmask" paradigm along with some local computation. The committees also make sure to update the accumulators of $\pi_{\text{MAC-check-hm}}$ and $\pi_{\text{mult-verify-hm}}$ along the way with each opening and multiplication, respectively. Finally, once all circuit layers have been computed, the final committees invoke the **Check State** phases of $\pi_{\text{MAC-check-hm}}$ and $\pi_{\text{mult-verify-hm}}$, then reconstruct the outputs to the clients. We note that, as it is remarked in the protocols of [Rachuri and Scholl 2022; Choudhuri et al. 2021], if the clients indeed have access to a broadcast channel in the last round of the protocol, or implement a broadcast over their point-to-point channels, then security with unanimous abort can be achieved by having the clients broadcast "abort", if their check on their output fails.

Figure 3.23: Protocol $\Pi_{\text{main-hm}}$

Input Phase: To form a SPDZ sharing of an input x_i possessed by $P_i \in C_{\text{clnt}}$:

1. P_i samples random degree-t_1 and degree-$2t_1$ Shamir sharings of x_i and distributes the corresponding shares to all parties in C_1.
2. C_1 then invokes $\mathcal{F}_{\text{double-rand-hm}}$ to get random double sharings of the global MAC key $([\Delta]^{C_1}_{t_1}, [\Delta]^{C_1}_{2t_1})$.
3. Next, parties in C_1 locally obtain authenticated sharing $[\![x_i]\!]^{C_1} \leftarrow ([x_i]^{C_1}_{2t_1}, [x_i]^{C_1}_{t_1} \cdot [\Delta]^{C_1}_{t_1}, [\Delta]^{C_1}_{2t_1})$ and invoke $\pi_{\text{ineff-reshare-hm}}$ on input $[\Delta]^{C_1}_{t_1}$ and $\pi_{\text{eff-reshare-hm}}$ on input $[\![x_i]\!]^{C_1}$.[a]
4. Parties in C_2 then invoke $\pi_{\text{get-rand-sharing}}$ to get $[\![r]\!]^{C_2}$ and invoke $\pi_{\text{mult-hm}}$ on input $[\![x_i]\!]^{C_2}$ and $[\![r]\!]^{C_2}$, as well as the identity gate procedure (below) on $[\![x_i]\!]^{C_2}$ so that C_4 gets $[\![x_i]\!]^{C_4}, [\![rx_i]\!]^{C_4}$.
5. Finally, C_4 invokes $\pi_{\text{mult-verify-hm}}$ on input (update, $\{([\![x_i]\!]^{C_4}, [\![rx_i]\!]^{C_4})\}_{i \in [|C_{\text{clnt}}|]}$), corresponding to each input.

Execution Phase:

1. Each Committee C_i of the execution phase will first of all invoke $\pi_{\text{ineff-reshare-hm}}$ on input $[\Delta]^{C_i}_{t_i}$ and $\pi_{\text{eff-reshare-hm}}$ on input $[\![r]\!]^{C_i}$ and $[\Delta]^{C_i}_{2t_i}$.
2. In parallel, every other committee (with the help of the others) will compute the gates at each layer of the circuit as below:

Addition: To perform addition on $[\![x]\!]^{C_i}$ and $[\![y]\!]^{C_i}$ (and identically for $[\![rx]\!]^{C_i}$ and $[\![ry]\!]^{C_i}$):

1. All parties in C_i agree on a special party P_{king} in C_{i+1} then invoke $\pi_{\text{get-rand-sharing}}$ to get $[\![s]\!]^{C_i}$.
2. Then, parties in C_i locally obtain $[\![x+y+s]\!]^{C_i}$ and open it to P_{king} while invoking $\pi_{\text{eff-reshare-hm}}$ on input $[\![s]\!]^{C_i}$.

3. While P_{king} distributes opened $x + y + s$ to the parties of C_{i+2}, all parties in C_{i+1} invoke $\pi_{\text{eff-reshare-hm}}$ on $[\![s]\!]^{C_{i+1}}$.

4. Parties in C_{i+2} finally locally compute $[\![x + y]\!]^{C_{i+2}} \leftarrow (x + y + s) - [\![s]\!]^{C_{i+2}}$.

5. Parties in C_{i+2} will also invoke $\pi_{\text{MAC-check-hm}}$ on input $(\text{update}, \{(x_m + y_m + s_m, [\Delta \cdot (x_m + y_m + s_m)]^{C_{i+2}}_{2t_{i+2}})\}_{m \in [T]})$ corresponding to all of the openings of the above form they receive for the addition gates at this layer of the circuit.[b]

Identity Gates: C_i forwards $[\![x]\!]^{C_i}$, $[\![rx]\!]^{C_i}$ to C_{i+2} in a similar fashion as addition above.

Multiplication: To multiply $[\![x]\!]^{C_i}$ and $[\![y]\!]^{C_i}$, invoke $\pi_{\text{mult-hm}}$ on them (and identically for $[\![rx]\!]^{C_i}$ and $[\![y]\!]^{C_i}$). Then invoke $\pi_{\text{mult-verify-hm}}$ on input $(\text{update}, \{([\![x_m y_m]\!]^{C_{i+2}}, [\![(rx)_m y_m]\!]^{C_{i+2}})\}_{m \in [T_i]})$, corresponding to each multiplication performed at this layer of the circuit.

Output Phase:

1. Parties in the last committee C_ℓ who compute the shares of the output gates then invoke $\pi_{\text{MAC-check-hm}}$ on check. If it outputs Reject, then abort; else, continue.

2. Parties in $C_{\ell+2}$ (check of $\pi_{\text{MAC-check-hm}}$ takes 3 rounds) then invoke $\pi_{\text{mult-verify-hm}}$ on check. If it outputs Reject, then abort; else, continue.

3. Next, for Party P_j in $C_{\ell+9}$ (check of $\pi_{\text{mult-verify-hm}}$ takes 8 rounds), let $z^j, (\Delta \cdot z)^j$ be their respective shares of output wire $[z]^{C_{\ell+9}}_{2t_{\ell+9}}$ and MAC $[\Delta \cdot z]^{C_{\ell+9}}_{2t_{\ell+9}}$. Each P_j creates random degree-$t_{\ell+10}$ Shamir secret sharings $\left[z^j\right]^{C_{\ell+10}}_{t_{\ell+10}}, \left[(\Delta \cdot z)^j\right]^{C_{\ell+10}}_{t_{\ell+10}}$ and distributes the corresponding shares to the parties of $C_{\ell+10}$.

4. Then each party $P_l \in C_{\ell+10}$ computes $[z]^{C_{\ell+10}}_{t_{\ell+10}} \leftarrow \sum_{j \in C_{\ell+9}} c_j \cdot \left[z^j\right]^{C_{\ell+10}}_{t_{\ell+10}}$, and similarly for $[\Delta \cdot z]^{C_{\ell+10}}_{t_{\ell+10}}$, where c_j is the Lagrange reconstruction coefficient for a degree-$2t_{\ell+9}$ polynomial.

5. Finally, the parties of $C_{\ell+10}$ open the shares of each $[z]^{C_{\ell+10}}_{t_{\ell+10}}$, $[\Delta]^{C_{\ell+10}}_{t_{\ell+10}}$, and $[\Delta \cdot z]^{C_{\ell+10}}_{t_{\ell+10}}$ to the clients, who attempt to reconstruct them and check that indeed the product of the former two values equal the last value. If so, they output each z; else, they abort.

[a]Note that the computed MAC for $[\![x_i]\!]^{C_1}$ is re-randomized with a fresh 0-sharing in $\pi_{\text{eff-reshare-hm}}$.
[b]This invocation can be combined with that of the multiplication gates for this circuit layer.

Theorem 3.18. *Let $\mathcal{A}$ be an* R*-adaptive adversary in $\Pi_{\text{main-hm}}$. Then the protocol UC-securely computes $\mathcal{F}_{\text{DABB}}$ in the presence of $\mathcal{A}$ in the $(\mathcal{F}_{\text{rand-hm}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{double-rand-hm}}, \mathcal{F}_{\text{zero}})$-hybrid model.*

Proof. The proof of this Theorem follows very similarly to that of Theorem 3.9. We construct a Simulator ($\mathcal{S}$) that runs the adversary ($\mathcal{A}$) as a subroutine, and is given access to $\mathcal{F}_{\text{DABB}}$. It internally emulates the functionalities $\mathcal{F}_{\text{rand-hm}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{double-rand-hm}}, \mathcal{F}_{\text{zero}}$ and we implicitly assume that it passes all communication between $\mathcal{A}$ and the environment $\mathcal{Z}$. It keeps track of the current committee via inputs (Init, C) and $(\mathsf{Next\text{-}Committee}, C)$ from $\mathcal{F}_{\text{DABB}}$ (and therefore in which committees to simulate corresponding communication for circuit gates). The simulator uses bad, initially set to 0, to detect any bad behavior from $\mathcal{A}$. If so, it sets bad $\leftarrow 1$. The simulation proceeds as follows:

Input: On input $(\mathsf{Input}, \mathsf{id}_x)$, if it is from an honest party, simulate the double sharings using random values. From the security of Shamir secret sharing, this is a perfect simulation. For inputs from all (even adversarial) parties, simulate $\pi_{\text{ineff-reshare-hm}}$ as in Lemma 3.14 and $\pi_{\text{eff-reshare-hm}}$ as in Lemma 3.13. If $\mathcal{A}$ cheats when resharing a value (i.e., by sending a wrong share), set bad $\leftarrow 1$. Also, simulate the multiplication as below.

Addition (and similarly for identity gates): On input $(\mathsf{Add}, \mathsf{id}_z, \mathsf{id}_x, \mathsf{id}_y)$, simulate $\pi_{\text{eff-reshare-hm}}$ as in Lemma 3.13, and $\pi_{\text{MAC-check-hm}}$ as in Lemma 3.15. Additionally, simulate the opening of $[\![x + y + s]\!]^{C_i}$ with random values. Since s is uniformly random and unknown to $\mathcal{A}$, this is a perfect simulation. If $\mathcal{A}$ cheats when sending some universal hash value during $\pi_{\text{MAC-check-hm}}$, abort. If $\mathcal{A}$ cheats either when resharing or opening a value (i.e., by sending a wrong share), set bad $\leftarrow 1$.

Multiplication: On input $(\mathsf{Mult}, \mathsf{id}_z, \mathsf{id}_x, \mathsf{id}_y)$, simulate $\pi_{\text{mult-hm}}$ as in Lemma 3.17 and $\pi_{\text{mult-verify-hm}}$ as in Lemma 3.16. If $\mathcal{A}$ cheats when sending some universal hash value during $\pi_{\text{MAC-check-hm}}$, abort.

If for any c part of a multiplication triple, the additive error $\delta_c \neq 0$, set bad $\leftarrow 1$. Additionally, if $\mathcal{A}$ cheats either when resharing or opening a value (i.e., by sending a wrong share), set bad $\leftarrow 1$.

Output: In the **Check State** phase of $\pi_{\text{MAC-check-hm}}$, for the distributed degree t_{i+1} Shamir sharings we know from Lemma 3.13, that at least one share of $[\sigma]_{2t_i}^{C_i}$ is uniformly random and unknown to $\mathcal{A}$, while all others can be computed by $\mathcal{A}$. For those that are uniformly random, by the security of Shamir secret sharings, the distributed degree t_{i+1} shares can be simulated with random values. For those that are known, the simulator can simply sample the distributed degree t_{i+1} shares on its own. Now, if $\mathcal{S}$ ever set bad $\leftarrow 1$ because $\mathcal{A}$ cheated when opening or resharing a value, $\mathcal{S}$ sends random values for σ on behalf of the honest parties, then aborts. Otherwise, $\mathcal{S}$, samples random shares for the honest parties such that they reconstruct to 0 and are consistent with the degree t_{i+1} sampled by the simulator above, and sends them to $\mathcal{A}$. If $\mathcal{A}$ cheats during the *check state* phase of $\pi_{\text{MAC-check-hm}}$ (e.g., by distributing incorrect degree-t_i shares of some share σ^i), $\mathcal{S}$ aborts. In the *check state* phase of $\pi_{\text{mult-verify-hm}}$, $\mathcal{S}$ sends random shares on behalf of the honest parties for the opening of r. If $\mathcal{A}$ cheats by opening the wrong values for r, $\mathcal{S}$ aborts after the MAC check (as above). If $\mathcal{S}$ ever set bad $\leftarrow 1$ because $\mathcal{A}$ added non-zero error to the c part of a multiplication triple, $\mathcal{S}$ sends random values on behalf of the honest parties for $(u - r \cdot w)$, then aborts. Otherwise, $\mathcal{S}$ records the values sent by $\mathcal{A}$ for $(u - r \cdot w)$, then samples shares such that $(u - r \cdot w) = 0$, and sends them to $\mathcal{A}$. If $\mathcal{A}$ cheats when opening $(u - r \cdot w)$, $\mathcal{S}$ aborts after the MAC check (as above).

Finally, $\mathcal{S}$ gets the outputs from $\mathcal{F}_{\text{DABB}}$ and forwards it to $\mathcal{A}$. $\mathcal{S}$ then forwards whatever it receives from $\mathcal{A}$ back to $\mathcal{F}_{\text{DABB}}$. We can simulate the opening of the output wires (and MACs) in a similar fashion as the *check state* phase of $\pi_{\text{MAC-check-hm}}$.

From all of the Lemmas, we have that the simulation is perfect up until the output phase. By Lemmas 3.15 (a similar argument holds for checking the MACs of the output wires) and 3.16, $\mathcal{A}$ is only able to cheat in the real world with probability negligible in p. Thus, the distance between the real-world and the simulation is negligible in p.

□

3.5 Two-Thirds Honest Majority Protocol

3.5.1 Technical Overview

Let $C_1, C_2, \ldots$ be the committees involved, each having n parties and t corruptions, with $n = 3t + 1$. In this chapter, we will mostly work with degree $d = 2t$ for Shamir secret sharings $[x]_d^{C_i}$. As previous works in Fluid MPC, we assume that the circuit is *layered*, meaning that the multiplication gates in a given layer can only depend on the outputs of the layer inmediately before it. The resulting circuit has addition, multiplication, and *identity* gates for relaying values from one layer to the next one. First each client, who provides input $x \in \mathbb{F}$ to the computation, secret-shares their input towards the first committee C_1 as $[x]_{2t}^{C_1}$. At this point, the parties in C_1 have sharings of the first layer of the computation, and the goal now is to let them evaluate the first layer of the circuit so that parties in a future committee get sharings of the outputs of this layer. This is then continued for each layer, until certain committee obtains shares of the output layer, which can be reconstructed to the clients. From this general template, the core question becomes the following: design a protocol so that, starting from a committee C_i holding shares of two values $[x]_{2t}^{C_i}$ and $[y]_{2t}^{C_i}$, a future committee C_j with $j > i$ can learn the product $[xy]_{2t}^{C_{i+1}}$. Such protocol enables processing multiplication gates, and our overview focuses mostly on illustrating how we do this. Handling identity gates (and in fact also addition gates) boils down to one committee C_i holding $[x]_{2t}^{C_i}$, transfering this to C_j so that they obtain $[x]_{2t}^{C_j}$. This is a strictly easier task than multiplication and it is approached via a simplification of our multiplication protocol.

3.5.1.1 Challenges of our multiplication protocol.

For notational convenience let us relabel the committee who holds the initial sharings from C_i to C_{i+1}, that is, committee C_{i+1} has two sharings $[x]_{2t}^{C_{i+1}}$ and $[y]_{2t}^{C_{i+1}}$, and the goal is to multiply each x with y. The general structure of our protocol is the following. At a high level, the idea is to let committee C_{i+1} use a multiplication triple $([a]_{2t}^{C_{i+1}}, [b]_{2t}^{C_{i+1}}, [c]_{2t}^{C_{i+1}})$ where $c = ab$ to compute the product between x and y, as standard in Beaver-based multiplication. This consists of first computing locally the sharings $[d]_{2t}^{C_{i+1}} = [x]_{2t}^{C_{i+1}} + [a]_{2t}^{C_{i+1}}$ and $[e]_{2t}^{C_{i+1}} = [y]_{2t}^{C_{i+1}} + [b]_{2t}^{C_{i+1}}$, reconstructing d and e, and taking the linear combination $[xy]_{2t}^{*} = de - e\,[a]_{2t}^{*} - d\,[b]_{2t}^{*} + [c]_{2t}^{*}$. However, there are two complications with this approach. The first is that, because we are in the maximal fluidity setting, reconstruction cannot be done among the members of C_{i+1} themselves, but rather, towards the next committee. To make matters worse, reconstructing in one round (*i.e.* towards the immediate next committee C_{i+2}) would involve *quadratic* communication (since this requires every party in C_{i+1} to send a share to every party in C_{i+2}), which we cannot afford. To obtain linear communication we make use of the "multiple king idea" from [Damgård and Nielsen 2007] (explained in more detail later in the section), but this results in committee C_{i+3} learning the reconstructions of d and e, not C_{i+2}. The second complication arises from the fact that the linear combination leading to xy described above must be computed by the committee that knows d and e, C_{i+3}, but this committee must also have a triple $([a]_{2t}^{C_{i+3}}, [b]_{2t}^{C_{i+3}}, [c]_{2t}^{C_{i+3}})$, which is currently held by committee C_{i+1} (this is why we put asterisks $*$ in the sharings when describing the linear combination). Solving this issue requires designing a method for committee C_{i+1} to transfer sharings to committee C_{i+3}.

Finally, another aspect we have overlooked is the generation of the multiplication triple: committee C_{i+1} needs to obtain $([a]_{2t}^{C_{i+1}}, [b]_{2t}^{C_{i+1}}, [c]_{2t}^{C_{i+1}})$ in a first place. For this, we will let committee C_{i+1} first get $[a]_{t}^{C_{i+1}}$ and $[b]_{t}^{C_{i+1}}$ (notice the degree t instead of $2t$), from which the parties in C_{i+1} can derive *locally* $[ab]_{2t}^{C_{i+1}} = [a]_{t}^{C_{i+1}} \cdot [b]_{t}^{C_{i+1}}$. Since any degree-$t$ sharing is also a degree-$2t$ sharing, the parties in C_{i+1} interpret $[a]_t^{C_{i+1}}$ as $[a]_{2t}^{C_{i+1}}$ (and similarly for b). This way, the parties have obtained the required triple. An avid reader may note that the computed $[ab]_{2t}^{C_{i+1}}$ is *not* a random degree-$2t$ sharing of $c = a \cdot b$, since the underlying polynomial is not random but rather the product of two degree-t polynomials. Also, $[a]_{2t}^{C_{i+1}}$ (and same for b) is not a random degree-$2t$ sharing, since the underlying polynomial has degree t. However, as we will see, this turns out to not be a problem, and the randomization involved when re-sharing helps us prevent leakage from this underlying structure.

One final question left is how committee C_{i+1} gets $[a]_t^{C_{i+1}}$ (and $[b]_t^{C_{i+1}}$) in a first place. For this, we use committee C_i: the parties in C_i execute the random sharing generation protocol from [Beerliová-Trubíniová and Hirt 2008], except that the receivers are the parties in C_{i+1}. We provide more details below.

3.5.1.2 Parties in C_i generate random sharings towards C_{i+1}.

Our multiplication protocol operates in *batches*: $t + 1$ products are handled simultaneously. Committee C_{i+1} has multiple sharings $[x_1]_{2t}^{C_{i+1}}, \dots, [x_{t+1}]_{2t}^{C_{i+1}}$ and $[y_1]_{2t}^{C_{i+1}}, \dots, [y_{t+1}]_{2t}^{C_{i+1}}$, and the goal is to multiply each x_α with y_α for $\alpha \in [t + 1]$.

For the multiplication, committee C_{i+1} needs random sharings $\{[a_\alpha]_t^{C_{i+1}}, [b_\alpha]_t^{C_{i+1}}\}_{\alpha \in [t+1]}$ which will be used for producing multiplication triples. We use the standard approach from [Beerliová-Trubíniová and Hirt 2008] for this, in which each party samples and distributes some random sharings, which are then combined with an appropriate matrix to obtain truly random sharings. However, since parties in C_{i+1} cannot communicate with each other, they must receive these sharings from C_i. In more detail, to generate a set of *t+1* random sharings $[a_\alpha]_t^{C_{i+1}}$ for $\alpha \in [t + 1]$, each party $P_j \in C_i$ samples a random $s_j \in \mathbb{F}$ and distributes $\left[s_j\right]_t^{C_{i+1}}$ towards C_{i+1}.

The next step is for the parties in C_{i+1} to *locally* apply a hyper-invertible matrix to the vector $([s_1]_t^{C_{i+1}}, \dots, [s_n]_t^{C_{i+1}})$, obtaining $([a_1]_t^{C_{i+1}}, \dots, [a_n]_t^{C_{i+1}})$. These matrices, defined in [Beerliová-Trubíniová and Hirt 2008], have several important properties that simultaneously help with randomness extraction and cheating verification, which is used to handle the fact that the parties in C_i may distribute incorrect random sharings (*e.g.* using incorrect degree); but for this a check similar to [Beerliová-Trubíniová and Hirt 2008] is performed. The sharings that the parties in C_{i+1} will finally use are the first $t + 1$ sharings $([a_1]_t^{C_{i+1}}, \dots, [a_{t+1}]_t^{C_{i+1}})$, and $([b_1]_t^{C_{i+1}}, \dots, [b_{t+1}]_t^{C_{i+1}})$ produced in a similar way.

3.5.1.3 Parties in C_{i+1} use the random sharings.

These sharings are used to obtain Beaver triples: $[a_\alpha]_t^{C_{i+1}}$ and $[b_\alpha]_t^{C_{i+1}}$ can be *locally* multiplied to obtain $[c_\alpha]_{2t}^{C_{i+1}}$. From now on $[a_\alpha]_t^{C_{i+1}}$ and $[b_\alpha]_t^{C_{i+1}}$ are interpreted as $[a_\alpha]_{2t}^{C_{i+1}}$ and $[b_\alpha]_{2t}^{C_{i+1}}$, respectively. Then the parties in C_{i+1} execute the first step of Beaver-based multiplication: they add locally $[d_\alpha]_{2t}^{C_{i+1}} \leftarrow [x_\alpha]_{2t}^{C_{i+1}} + [a_\alpha]_{2t}^{C_{i+1}}$ and $[e_\alpha]_{2t}^{C_{i+1}} \leftarrow [y_\alpha]_{2t}^{C_{i+1}} + [b_\alpha]_{2t}^{C_{i+1}}$, and then the goal is to reconstruct these values. Of course, since the members in committee C_{i+1} cannot talk to each other in the maximal fluidity setting, these parties would reconstruct these values towards committee C_{i+2}. This is done via the standard efficient reconstruction procedure from [Damgård and Nielsen 2007], which consists of the parties first expanding the sharings $\{[d_\alpha]_{2t}^{C_{i+1}}\}_{\alpha\in[t+1]}$ with an error correcting code, instantiated by multiplication with a super-invertible matrix, to obtain $\{\left[d'_\beta\right]_{2t}^{C_{i+1}}\}_{\beta\in[n]}$ (and similarly $\{\left[e'_\beta\right]_{2t}^{C_{i+1}}\}_{\beta\in[n]}$); this is followed by all parties in C_{i+1} sending their shares of $[d_k]_{2t}^{C_{i+1}}$ and $[e_k]_{2t}^{C_{i+1}}$ to each $P_k \in C_{i+2}$.

At this point the parties in C_{i+2} have "shares" $(d'_1, \dots, d'_n)$ and $(e'_1, \dots, e'_n)$, which reconstruct to polynomials encoding $(d_1, \dots, d_{t+1})$ and $(e_1, \dots, e_{t+1})$ respectively. These need to be "reconstructed", so these "sharings" are sent to committee C_{i+3}, who learn $(d_1, \dots, d_{t+1})$ and $(e_1, \dots, e_{t+1})$. Committee C_{i+3} will execute Beaver-based multiplication, which works by taking the local linear combination

$$[x_\alpha y_\alpha]_{2t}^{C_{i+3}} = d_\alpha e_\alpha - d_\alpha [b_\alpha]_{2t}^{C_{i+3}} - e_\alpha [a_\alpha]_{2t}^{C_{i+3}} + [c_\alpha]_{2t}^{C_{i+3}} . \tag{3.1}$$

However, in order to do this the parties in C_{i+3} need the sharings of the triple $([a_\alpha]_{2t}^{C_{i+3}}, [b_\alpha]_{2t}^{C_{i+3}},$

$[c_\alpha]_{2t}^{C_{i+3}}$).

3.5.1.4 Committee C_{i+3} gets the triple - Resharing protocol based on packed secret sharing.

Recall that the parties in C_{i+1} have $([a_\alpha]_{2t}^{C_{i+1}}, [b_\alpha]_{2t}^{C_{i+1}}, [c_\alpha]_{2t}^{C_{i+1}})$. These sharings will be transferred to committee C_{i+3} via a resharing protocol. This is done in two steps where parties in C_{i+1} send packed secret sharings of their shares. Next, the parties in C_{i+2} can combine these packed sharings of shares into one packed sharing and then Shamir sharings of their packed shares are sent to C_{i+3} to get non-packed shares of the original secrets. See details below.

Committee C_{i+2} gets $([\mathbf{a}]_{2t}^{C_{i+2}}, [\mathbf{b}]_{2t}^{C_{i+2}}, [\mathbf{c}]_{2t}^{C_{i+2}})$. For committee C_{i+1} to send the triples to committee C_{i+3}, these sharings have to pass through C_{i+2} first. Towards this, parties in C_{i+1} transfer these sharings to C_{i+2}, but they do so with *packed secret-sharing*, which is crucial for efficiency. We focus on how to transfer $\{[a_\alpha]_{2t}^{C_{i+1}}\}$ so that committee C_{i+3} gets $[\mathbf{a}]_{2t}^{C_{i+2}}$, with the other sharings handled in the same way.

Let us denote the j-th share of each $[a_\alpha]_{2t}^{C_{i+1}}$ by a_α^j, which satisfy $\sum_{j=1}^{2t+1} L_j(0)a_\alpha^j = a_\alpha$, where $L_j(\mathsf{X})$ are the Lagrange polynomials. Party $P_j \in C_{i+1}$, having $\mathbf{a}_{[1,t+1]}^j = (a_1^j, \ldots, a_{t+1}^j)$, distributes sharings $\left[\mathbf{a}_{[1,t+1]}^j\right]_{2t}^{C_{i+2}}$ to committee C_{i+2}. At this point each party in this committee can compute locally

$$\sum_{j=1}^{2t+1} L_j(0)\left[\mathbf{a}_{[1,t+1]}^j\right]_{2t}^{C_{i+2}} = \left[\sum_{j=1}^{2t+1} L_j(0)\cdot \mathbf{a}_{[1,t+1]}^j\right]_{2t}^{C_{i+2}} = [\mathbf{a}]_{2t}^{C_{i+2}},$$

where $\mathbf{a} = (a_1, \ldots, a_{t+1})$. Notice that a corrupt party $P_j \in C_{i+1}$ may distribute $\left[\mathbf{a}_{[1,t+1]}^j\right]_{2t}^{C_{i+2}}$ incorrectly. For example, the underlying secret may not correspond to $\mathbf{a}_{[1,t+1]}^j$. We discuss how to address this later in the section but for now, let us hint that this is prevented by leveraging the fact that each row of the "matrix" $[\mathbf{a}_{[1,t+1]}^1 \| \cdots \| \mathbf{a}_{[1,t+1]}^n]$ has to be *consistent* with a polynomial of degree $\leq 2t$, and this bounds the adversary to use the correct secrets.

Committee C_{i+3} obtains $([a_\alpha]_{2t}^{C_{i+3}}, [b_\alpha]_{2t}^{C_{i+3}}, [c_\alpha]_{2t}^{C_{i+3}})$. Here, the sharings $([\mathbf{a}]_{2t}^{C_{i+2}}, [\mathbf{b}]_{2t}^{C_{i+2}}, [\mathbf{c}]_{2t}^{C_{i+2}})$ held by committee C_{i+2} are "unpacked" towards committee C_{i+3}. As before, we focus on $[\mathbf{a}]_{2t}^{C_{i+2}}$ for the sake of exposition. Let us denote the share of party $P_j \in C_{i+2}$ of $[\mathbf{a}]_{2t}^{C_{i+2}}$ by $\mathbf{a}^j \in \mathbb{F}$, which satisfy $a_\alpha = \sum_{j=1}^{2t+1} L_j(-\alpha) \cdot \mathbf{a}^j$. Each of these parties secret-shares their share as $\left[\mathbf{a}^j\right]_{2t}^{C_{i+3}}$ towards committee C_{i+3}. Due to the observation above, we have that

$$\sum_{j=1}^{2t+1} L_j(-\alpha) \cdot \left[\mathbf{a}^j\right]_{2t}^{C_{i+3}} = \left[\sum_{j=1}^{2t+1} L_j(-\alpha) \cdot \mathbf{a}^j\right]_{2t}^{C_{i+3}} = [a_\alpha]_{2t}^{C_{i+3}},$$

so the parties in C_{i+3} can obtain the desired shares. A similar approach is followed to obtain $[b_\alpha]_{2t}^{C_{i+3}}$ and $[c_\alpha]_{2t}^{C_{i+3}}$.

3.5.1.5 Achieving active security.

The protocol sketched so far can be attacked by an active adversary at the following places:

1. The generation of the random sharings by committee C_i may be done inconsistently.
2. The parties in C_{i+1} may send incorrect shares of $\left[d'_k\right]_{2t}^{C_{i+1}}$ or $\left[e'_k\right]_{2t}^{C_{i+1}}$ to some $P_k \in C_{i+2}$, or P_k may send an incorrect d'_k to C_{i+3}.
3. A corrupt party $P_j \in C_{i+1}$ may reshare $\left[\mathbf{a}^j_{[1,t+1]}\right]_{2t}^{C_{i+2}}$ (or $\left[\mathbf{b}^j_{[1,t+1]}\right]_{2t}^{C_{i+2}}$, or $\left[\mathbf{c}^j_{[1,t+1]}\right]_{2t}^{C_{i+2}}$) inconsistently to committee C_{i+2}.
4. A corrupt party $P_j \in C_{i+2}$ may reshare $\left[\mathbf{a}^j\right]_{2t}^{C_{i+3}}$ (or $\left[\mathbf{b}^j\right]_{2t}^{C_{i+3}}$, or $\left[\mathbf{c}^j\right]_{2t}^{C_{i+3}}$) inconsistently to committee C_{i+3}.

The first item is addressed by using the hyper-invertible matrix verification from [Beerliová-Trubíniová and Hirt 2008], where some of the mapped sharings $([a_1]_{t,2t}^{C_{i+1}}, \dots, [a_n]_{t,2t}^{C_{i+1}})$, held by the receiving C_{i+1}, are opened. In our case, these are opened towards committee C_{i+2} who performs the check in [Beerliová-Trubíniová and Hirt 2008]. Since this follows relatively straightforwardly from [Beerliová-Trubíniová and Hirt 2008], we do not provide details in this overview, and refer the reader to Section 3.5.2.1 where we present our random sharings protocol. On the other hand, the second item is easily handled by performing error detection.

Items (3) and (4) require more care, and we provide an overview on these below.

Ensuring consistency of $\left[\mathbf{a}^j_{[1,t+1]}\right]^{C_{i+2}}_{2t}$. A party $P_j \in C_{i+1}$ may cheat by sending $\left[\mathbf{a}^j_{[1,t+1]} + \delta^j\right]^{C_{i+2}}_{2t}$, where $\delta^j \neq \mathbf{0}$.[10] First, let $\mathbf{H}$ be the $(n-2t-1)\times n$ matrix such that $\mathbf{H}\cdot\mathbf{x} = \mathbf{0}$ if and only if $\mathbf{x} \in \mathbb{F}^n$ are valid shares of a polynomial of degree $\leq 2t$. The intuitive idea behind our check is simple. Let $\mathbf{A}$ be the $n\times(t+1)$ matrix whose j-th row is $\mathbf{a}^j_{[1,t+1]}$. We have that for any index $\alpha \in [t+1]$, the α-th column is a degree-$2t$ sharings. This means the matrix satisfies $\mathbf{H}\cdot\mathbf{A} = \mathbf{0} \in (n-2t-1)\times(t+1)$. Let $\mathbf{D}$ be the $n\times(t+1)$ matrix whose j-th row is δ^j (we define $\delta^{\mathrm{j}} = \mathbf{0}$ for an honest party $P_j \in C_{i+1}$). Since the parties in C_{i+2} *allegedly* have sharings of each row $\left[\mathbf{a}^j_{[1,t+1]} + \delta^j\right]^{C_{i+2}}_{2t}$ of $\mathbf{A}+\mathbf{D}$, the parties can *locally* compute shares of each row of $\mathbf{H}\cdot(\mathbf{A}+\mathbf{D}) = \mathbf{H}\cdot\mathbf{A} + \mathbf{H}\cdot\mathbf{D} = \mathbf{H}\cdot\mathbf{D}$. Then the parties in C_{i+2} reconstruct these sharings towards committee C_{i+3}, and check that the corresponding secrets are all $\mathbf{0}$. It is possible to verify that, given that $\mathbf{D}$ only contains at most t non-zero rows, the only way in which $\mathbf{H}\cdot\mathbf{D}$ can equal zero is if $\mathbf{D} = \mathbf{0}$, so this check indeed ensures that no cheating occurred.

A detail we have neglected is that, simply reconstructing all rows of $\mathbf{H}\cdot\mathbf{D}$ towards all members of C_{i+3} is too expensive since it requires n^2 communication. Instead, the parties in C_{i+2} apply a hyper-invertible matrix that maps these $n-2t-1$ rows to n rows, and the j-th row is reconstructed only towards $P_j \in C_{i+3}$. This ensures that if there is at least one row in $\mathbf{H}\cdot\mathbf{D}$ that is not zero, then at least one *honest* party in C_{i+3} receives a non-zero row. This party signals this to the parties in C_{i+4}, who learn at that point whether there was an error in the original sharings or not.

The downside of this approach is that some honest parties in C_{i+3} may not know yet that an

[10]The "worst-case degree" for an inconsistent sharing is $2t$, since the shares of the $n-t = 2t+1$ honest parties uniquely define a polynomial of degree $\leq 2t$. Hence, a corrupt party cannot cheat on the degree, and at worst can cheat by changing the secret

error took place. This is an important and subtle issue, since some honest parties in C_{i+3} may not know yet that some $\left[\mathbf{a}^j_{[1,t+1]}\right]_{2t}^{C_{i+2}}$ is inconsistent, and yet they would proceed with the protocol specification. This could be a problem if the steps that follows leak sensitive information when they are executed on inconsistent shares, as noted for example in [Goyal et al. 2019]. However, this is not a problem in our case. At a high level, compared to [Goyal et al. 2019], we use degree-$2t$, while they use degree-t (which are important for G.O.D.). This removes all "extra redundancy" that the honest parties' shares may carry, which is what causes privacy issues in [Goyal et al. 2019].

Ensuring consistency of $\left[\mathbf{a}^j\right]_{2t}^{C_{i+3}}$. Recall that each $\mathbf{a}^j \in \mathbb{F}$ is a share of the degree-$2t$ packed sharing $[\mathbf{a}]_{2t}^{C_{i+2}}$. As it turns out, the consistency of $\left[\mathbf{a}^j\right]_{2t}^{C_{i+3}}$ is verified in a similar manner as above. This is thanks to the observation that $(\mathbf{a}^1, \ldots, \mathbf{a}^n)$ is a consistent degree-$2t$ sharing (it equals $[\mathbf{a}]_{2t}^{C_{i+2}}$), so it should equal $\mathbf{0}$ when multiplied by the matrix $\mathbf{H}$. Furthermore, since the parties in C_{i+3} have sharings of each $\left[\mathbf{a}^j\right]_{2t}^{C_{i+3}}$, they can compute shares of this matrix product. As before, instead of sending these shares to all parties in C_{i+4}, the parties in C_{i+3} first apply a super-invertible matrix and reconstruct the j-th entry to the j-th party in C_{i+4}, who checks the underlying secret is 0. If one party finds an error, this is reported to all parties in C_{i+5}.

3.5.1.6 Final remarks

Communication complexity. Notice that communication of all the steps described above involves each party from one committee sending a constant amount of field elements to the parties in the next committee. This leads to a communication of $O(n^2)$. However, recall that this is to process a batch of $t + 1 = \Omega(n)$ multiplication gates, which means that the amortized cost per gate is $O(n)$, as desired.

Input and addition gates. Finally, we point out that similar techniques as sketched above are used to handle input and addition gates: the parties somehow reshare the values, and some checks are performed to ensure consistency. This is done as the resharing of the triples, essentially.

3.5.2 Formal Protocol

3.5.2.1 Basic Functionalities

This section discusses three crucial functionalities for our final protocol, whose instantiations are somewhat direct adaptations of previous non-fluid works. The first is a functionality for generating sharings of uniformly random values, which recall from the overview in Section 3.5.1 is one of the ingredients needed to generate multiplication triples, which are used to process multiplication gates. The other two functionalities are concerned with how the clients provide inputs, and how they get outputs once the computation is done.

Random Sharings Generation [Beerliová-Trubíniová and Hirt 2008] We define the functionality for random sharing generation $\mathcal{F}_{\text{rand-2/3-hm}}$ below. It first generates random sharings $[r_1]_t^{C_{i+1}}, \ldots, [r_{t+1}]_t^{C_{i+1}}$ consistent with t shares received from the adversary corresponding to corrupted parties for each. It then distributes to the honest parties of committee C_{i+1} their shares of these sharings, summed with their shares of error-sharings $[e_1]_{t'}^{C_{i+1}}, \ldots, [e_{t+1}]_{t'}^{C_{i+1}}$, respectively, received from the adversary, where $t' \leq 2t$. Finally, if any such sharing $[e_1]_{t'}^{C_{i+1}}$ is not equal to the all-0 sharing, then $\mathcal{F}_{\text{rand-2/3-hm}}$ sends abort to the honest parties of C_{i+3}.

Figure 3.24: Functionality $\mathcal{F}_{\text{rand-2/3-hm}}$

1. $\mathcal{F}_{\text{rand-2/3-hm}}$ receives from the adversary the set of shares $\left(\{r_1^k\}_{k\in\mathcal{T}_{C_{i+1}}}, \ldots, \{r_{t+1}^k,\}_{k\in\mathcal{T}_{C_{i+1}}}\right)$ and sharings $([e_1]_{t'}^{C_{i+1}}, \ldots, [e_{t+1}]_{t'}^{C_{i+1}})$, where $t' \leq 2t$.

2. $\mathcal{F}_{\text{rand-2/3-hm}}$ then samples $r_1, r_2, \ldots, r_{n-2t}$ randomly, and generates $[r_1]_t^{C_{i+1}}, \ldots, [r_{t+1}]_t^{C_{i+1}}$ such that for every $j \in [t+1]$ and $k \in \mathcal{T}_{C_{i+1}}$, the k-th shares of $\left[r_j\right]_t^{C_{i+1}}$ are r_j^k.

3. Next, for every $j \in [t+1]$, $l \in \mathcal{H}_{C_{i+1}}$, $\mathcal{F}_{\text{rand-2/3-hm}}$ sends the l-th shares of $\left[r_j\right]_t^{C_{i+1}} + \left[e_j\right]_{t'}^{C_{i+1}}$ to P_l.

4. Finally, if any $\left[e_j\right]_{t'}^{C_{i+1}}$ is not equal to $[0]_0^{C_{i+1}}$, i.e., the all-0 sharing, then $\mathcal{F}_{\text{rand-2/3-hm}}$ sends abort to the honest parties of C_{i+3}. Otherwise, $\mathcal{F}_{\text{rand-2/3-hm}}$ asks the adversary whether to continue, and if the adversary replies (abort, A) for $A \subseteq \mathcal{H}_{C_{i+3}}$, then $\mathcal{F}_{\text{rand-2/3-hm}}$ sends abort to the honest parties A of C_{i+3}.

Functionality $\mathcal{F}_{\text{rand-2/3-hm}}$ is instantiated by Protocol $\Pi_{\text{rand-2/3-hm}}$ below. As we discussed in the overview, this is an adaptation of the random sharing generation protocol by [Damgård and Nielsen 2007], except messages are sent from one committee to the next.

Figure 3.25: Protocol $\Pi_{\text{rand-2/3-hm}}$

Usage: With help from committee C_i, committee C_{i+1} outputs $t+1$ random sharings $[r]_t^{C_{i+1}}$, and uses committees C_{i+2} and C_{i+3} to ensure their t-consistency.

1. All parties P_j in C_i sample random s_j and share it to the parties of C_{i+1} using degree t.

2. The parties of C_{i+1} first locally apply $(n \times n)$ hyper-invertible matrix $\mathbf{M}$ to the sharings from C_i to obtain

$$([r_1]_t^{C_{i+1}}, \ldots, [r_n]_t^{C_{i+1}})^\intercal \leftarrow \mathbf{M} \cdot ([s_1]_t^{C_{i+1}}, \ldots, [s_n]_t^{C_{i+1}})^\intercal.$$

3. While the Verification Phase (below) is executed, committee C_{i+1} outputs random sharings $([r_1]_t^{C_{i+1}}, \ldots, [r_{t+1}]_t^{C_{i+1}})$.

Verification Phase:

1. Then, the parties of C_{i+1} open the last $2t$ sharings $[r_{t+2}]_t^{C_{i+1}}, \ldots, [r_n]_t^{C_{i+1}}$ to the last $2t$ parties of C_{i+2}, respectively.

2. The last $2t$ parties P_l of C_{i+2} then check that indeed $[r_l]_t^{C_{i+1}}$ is t-consistent.

3. If this check fails for some P_j, they send abort to the parties of C_{i+3}.

The communication complexity of $\Pi_{\text{rand-2/3-hm}}$ is $n \cdot n = O(n^2)$ sharings in step 1, plus $2t \cdot n =$

$O(n^2)$ sharings from the verification, for a total of $O(n^2/(t+1)) = O(n)$ per random sharing.

Lemma 3.19. *Protocol* $\Pi_{\text{rand-2/3-hm}}$ *UC-realizes* $\mathcal{F}_{\text{rand-2/3-hm}}$.

Proof. Assume w.l.o.g., that the first $2t+1$ parties are honest. First we define the simulator $\mathcal{S}$:

1. $\mathcal{S}$ for $j \in [2t+1]$ first samples random sharings $\left[s_j\right]_t^{C_{i+1}}$, then sends the committee C_{i+1} corrupted parties' shares, $\{s_j^k\}_{k\in[2t+2,n]}$ to the adversary.
2. $\mathcal{S}$ then receives from the adversary $\{s_j^k\}_{k\in[2t+1]}$, for $j \in [2t+2, n]$, on behalf of the committee C_{i+1} honest parties.
3. For each $j \in [2t+2, n]$, $\mathcal{S}$ uses $s_j^1, \ldots, s_j^{t+1}$ to reconstruct $\left[s_j\right]_t^{C_{i+1}}$.
4. Then, for each $j \in [2t+2, n]$, letting $s_{j,t}^k$ be the k-th share of $\left[s_j\right]_t^{C_{i+1}}$ just reconstructed, $\mathcal{S}$ uses $(0, \ldots, 0, s_j^{t+2} - s_{j,t}^{t+2}, \ldots, s_j^{2t+1} - s_{j,t}^{2t+1})$ to reconstruct $\left[d_j\right]_{t'_j}^{C_{i+1}}$, where $t'_j \leq 2t$ is the smallest value that defines a consistent sharing with the above $2t+1$ shares.
5. $\mathcal{S}$ then computes

$$([r_1]_t^{C_{i+1}}, \ldots, [r_n]_t^{C_{i+1}})^\intercal \leftarrow \mathbf{M} \cdot ([s_1]_t^{C_{i+1}}, \ldots, [s_n]_t^{C_{i+1}})^\intercal$$

   and

$$([e_1]_{t'}^{C_{i+1}}, \ldots, [e_n]_{t'}^{C_{i+1}})^\intercal \leftarrow \mathbf{M} \cdot \left(0, \ldots, 0, [d_{2t+2}]_{t'_{2t+2}}^{C_{i+1}}, \ldots, [d_n]_{t'_n}^{C_{i+1}}\right)^\intercal$$

   on behalf of all of the honest parties of committee C_{i+1}, where $t' = \max\{t'_j\}_{j\in[2t+2,n]}$.
6. Using the $2t+1$ honest parties' shares of the sharings $([r_1]_t^{C_{i+1}}, \ldots, [r_{t+1}]_t^{C_{i+1}})$, $\mathcal{S}$ reconstructs the whole sharings, and sends the corrupted parties' shares to $\mathcal{F}_{\text{rand-2/3-hm}}$, along with $([e_1]_{t'}^{C_{i+1}}, \ldots, [e_{t+1}]_{t'}^{C_{i+1}})$.
7. Then for $j \in [2t+2, n]$, $\mathcal{S}$ sends to the corresponding corrupted party of committee C_{i+2}, the honest parties' shares of $\left[r_j\right]_t^{C_{i+1}} + \left[e_j\right]_{t'}^{C_{i+1}}$.

8. $\mathcal{S}$ next receives from the adversary, for $j \in [t+2, 2t+1]$, the corrupted parties' shares of $\left[r_j\right]_t^{C_{i+1}} + \left[e_j\right]_{t'}^{C_{i+1}}$, for the corresponding honest party of committee C_{i+2}. Together with the honest parties' shares of these sharings, $\mathcal{S}$ checks that $\left[r_j\right]_t^{C_{i+1}} + \left[e_j\right]_{t'}^{C_{i+1}}$ indeed defines a t-consistent sharing. If not, then $\mathcal{S}$ sends on behalf of P_j, abort to the corrupted parties of committee C_{i+3}.

9. Finally, if $\mathcal{S}$ sent any abort in the above step, it sets $A = \mathcal{H}_{C_{i+3}}$; otherwise it sets A to be those honest parties that receive from some corrupt party abort. Then, when $\mathcal{F}_{\text{rand-2/3-hm}}$ asks $\mathcal{S}$ whether to continue, it replies with (abort, A).

Now we argue that the real world and ideal world are distributed identically to the adversary. In Step 1 of $\Pi_{\text{rand-2/3-hm}}$, the adversary receives the t committee C_{i+1} corrupted parties' shares of the committee C_i honest parties' sharings $\left[s_j\right]_t^{C_{i+1}}$. Indeed, Step 1 of $\mathcal{S}$ is to sample random sharings in the same way for each honest party, and send the corrupted committee C_{i+1} parties their shares. Thus, the view of the adversary is identical in the real and ideal worlds for this step. For these degree-t sharings, there are $t+1$ total (random) degrees of freedom in defining the underlying polynomial, and thus the t shares that the adversary sees are uniformly random, while leaving 1 remaining (random) degree of freedom for each sharing, or $2t+1$ in total.

In Step 3 of $\Pi_{\text{rand-2/3-hm}}$, the honest parties output their shares of $([r_1]_{t'}^{C_{i+1}}, \ldots, [r_{t+1}]_{t'}^{C_{i+1}})$. Let us examine what honest parties output for their shares of the degree-t' sharings in the ideal world. In Step 3 of $\mathcal{S}$, for $j \in [2t+2, n]$, it uses the underlying first $t+1$ shares $s_j^1, \ldots, s_j^t$ received from the adversary to reconstruct $\left[s_j\right]_t^{C_{i+1}}$. Then, in Step 4, $\mathcal{S}$ uses the differences between all of the $2t+1$ shares $s_j^1, \ldots, s_j^{2t+1}$ received from the adversary and those defined by $\left[s_j\right]_t^{C_{i+1}}$ reconstructed above, to reconstruct sharings $\left[d_j\right]_{t'_j}^{C_{i+1}}$ (of course, for the first $t+1$, the difference will be 0). Next, in Step 5, $\mathcal{S}$ computes

$$([r_1]_t^{C_{i+1}}, \ldots, [r_n]_t^{C_{i+1}})^\intercal \leftarrow \mathbf{M} \cdot ([s_1]_t^{C_{i+1}}, \ldots, [s_n]_t^{C_{i+1}})^\intercal$$

and

$$([e_1]_{t'}^{C_{i+1}}, \dots, [e_n]_{t'}^{C_{i+1}})^\top \leftarrow \mathbf{M} \cdot \left(0, \dots, 0, [d_{2t+2}]_{t'_{2t+2}}^{C_{i+1}}, \dots, [d_n]_{t'_n}^{C_{i+1}}\right)^\top.$$

Finally, in Step 6, $\mathcal{S}$ sends to $\mathcal{F}_{\text{rand-2/3-hm}}$ the corrupted parties' shares of $([r_1]_t^{C_{i+1}}, \dots, [r_{t+1}]_t^{C_{i+1}})$, along with $([e_1]_{t'}^{C_{i+1}}, \dots, [e_{t+1}]_{t'}^{C_{i+1}})$. $\mathcal{F}_{\text{rand-2/3-hm}}$ then for each $j \in [t+1]$, samples random r_j and degree-t sharing consistent with this value and the t corrupted parties' shares of $\left[r_j\right]_t^{C_{i+1}}$ received from $\mathcal{S}$, reconstructs the corresponding sharing, and sends to the honest parties of C_{i+1} their shares of this sharing, plus their share of $\left[e_j\right]_{t'}^{C_{i+1}}$. Now, for each $k \in [2t+1]$, let us look at honest party P_k's computation of $(r_1^k, \dots, r_{t+1}^k)^\top \leftarrow \mathbf{M}_{\mathsf{r}([t+1])}^{\mathsf{c}([t+1])} \cdot (s_1^k \dots s_{t+1}^k)^\top + \mathbf{M}_{\mathsf{r}([t+1])}^{\mathsf{c}([t+2,n])} \cdot (s_{t+2}^k \dots s_n^k)^\top$. Since $\mathbf{M}$ is hyper-invertible, $\mathbf{M}_{\mathsf{r}([t+1])}^{\mathsf{c}([t+1])}$ is invertible, which means that the $t+1$ values r_j^k have a one-to-one correspondence with s_j^k, for $j \in [t+1]$. Since for each sharing $\left[s_j\right]_t^{C_{i+1}}, j \in [t+1]$ there is 1 remaining (random) degree of freedom, we can for the first honest party, conclude that r_j^1 is random. Since $\mathcal{F}_{\text{rand-2/3-hm}}$ only needs to sample 1 random value for each of the $t+1$ random sharings $([r_1]_t^{C_{i+1}}, \dots, [r_{t+1}]_t^{C_{i+1}})$, the real world and ideal world are distributed identically in determining these sharings. Note also that $[s_{t+2}]_t^{C_{i+1}}, \dots, [s_{2t+1}]_t^{C_{i+1}}$ still have 1 remaining (random) degrees of freedom, each, at this point. Furthermore, observe that for $k \in [2t+1], j \in [2t+2, n]$ the k-th share of $\left[s_j\right]_t^{C_{i+1}}$ added to that of $\left[d_j\right]_{t'_j}^{C_{i+1}}$, will be exactly that s_j^k received from the adversary. Moreover, by linearity, the k-th shares of $\mathbf{M} \cdot ([s_1]_t^{C_{i+1}}, \dots, [s_n]_t^{C_{i+1}})^\top + \mathbf{M} \cdot (0, \dots, 0, [d_{2t+2}]_{t'_j}^{C_{i+1}}, \dots, [d_n]_{t'_j}^{C_{i+1}})^\top$ will thus be exactly $\mathbf{M} \cdot (s_1^k, \dots, s_n^k)^\top$. Therefore, the values output by the honest parties in the ideal world are distributed identically to those output in the real world.

In Step 1 of the **Verification Phase** of $\Pi_{\text{rand-2/3-hm}}$, for each $j \in [2t+2, n]$, the adversary receives on behalf of the corresponding corrupted party of C_{i+2}: the honest parties' shares of $\left[r_j\right]_{t'}^{C_{i+1}}$. For these sharings, for $k \in [t+1]$, let us look at honest party P_k's computation of $(r_{2t+2}^k, \dots, r_n^k)^\top \leftarrow \mathbf{M}_{\mathsf{r}([2t+2,n])}^{\mathsf{c}([t+2,2t+1])} \cdot (s_{t+2}^k \dots s_{2t+1}^k)^\top + \mathbf{M}_{\mathsf{r}([2t+2,n])}^{\mathsf{c}([t+1])} \cdot (s_1^k \dots s_{t+1}^k)^\top + \mathbf{M}_{\mathsf{r}([2t+2,n])}^{\mathsf{c}([2t+2,n])} \cdot (s_{2t+2}^k \dots s_n^k)^\top$. Since $\mathbf{M}$ is hyper-invertible, $\mathbf{M}_{\mathsf{r}([2t+2,n])}^{\mathsf{c}([t+2,2t+1])}$ is invertible, which means that the t values $r_{2t+2}^k, \dots, r_n^k$ have a one-to-one correspondence with s_l^k, for $l \in [t+2, 2t+1]$. Since for each sharing $[s_l]_t^{C_{i+1}}, l \in [t+2, 2t+1]$

there is 1 remaining (random) degree of freedom, we can for $j \in [2t+2, n]$, conclude that the share of the first honest party r_j^1 is random. $\mathcal{S}$ in Step 7 computes and sends the simulated shares of honest parties of $\left[r_j\right]_{t'}^{C_{i+1}}$ by separating $\left[r_j\right]_{t'}^{C_{i+1}}$ into $\left[r_j\right]_{t}^{C_{i+1}} + \left[e_j\right]_{t'}^{C_{i+1}}$. For $\left[r_j\right]_{t}^{C_{i+1}}$, since the first honest party share is random using the same argument as above, and the last t are consistent with the first, and the adversary's t shares, the ideal world is identically distributed to the real world at this point.

When $\mathcal{S}$ receives from the adversary, for $j \in [t+2, 2t+1]$, the corrupted parties' shares of $\left[r_j\right]_{t'}^{C_{i+1}}$, it performs the same consistency check on behalf of the honest parties as they do in the real world. Thus, the distribution of abort messages from the honest parties of C_{i+2} to the corrupt parties of C_{i+3}, and whether the honest parties of C_{i+3} abort, are identical in the real and ideal worlds.

Finally, we need to show that if for some $j \in [t+1]$, $\left[e_j\right]_{t'}^{C_{i+1}} \neq [0]_0^{C_{i+1}}$ (i.e., the all-0 sharing) in the real world, then the honest parties of committee C_{i+3} abort (as they are forced to in the ideal world). We do so by proving the contrapositive; i.e., if the honest parties do not abort, then for every $j \in [t+1]$, $\left[e_j\right]_{t'}^{C_{i+1}} = [0]_0^{C_{i+1}}$. For this, we again use the power of hyper-invertible matrix $\mathbf{M}$. For each $k \in [n]$, party P_k of committee C_{i+1} computes

$$(r_{t+2}^k, \dots, r_{2t+1}^k)^\top \leftarrow \mathbf{M}_{\mathsf{r}([t+2,2t+1])}^{\mathsf{c}([2t+2,n])} \cdot (s_{2t+2}^k \dots s_n^k)^\top + \mathbf{M}_{\mathsf{r}([t+2,2t+1])}^{\mathsf{c}([2t+1])} \cdot (s_1^k \dots s_{2t+1}^k)^\top .$$

Since $\mathbf{M}$ is hyper-invertible, $\mathbf{M}_{\mathsf{r}([t+2,2t+1])}^{\mathsf{c}([2t+2,n])}$ is invertible, so we can write

$$(s_{2t+2}^k \dots s_n^k)^\top = \left(\mathbf{M}_{\mathsf{r}([t+2,2t+1])}^{\mathsf{c}([2t+2,n])}\right)^{-1} \cdot (r_{t+2}^k, \dots, r_{2t+1}^k)^\top - \left(\mathbf{M}_{\mathsf{r}([t+2,2t+1])}^{\mathsf{c}([2t+2,n])}\right)^{-1} \cdot \mathbf{M}_{\mathsf{r}([t+2,2t+1])}^{\mathsf{c}([2t+1])} \cdot (s_1^k \dots s_{2t+1}^k)^\top .$$

Now, since the honest parties did not abort, their checks passed, which means that for $j \in [t+2, 2t+1]$, the shares $r_j^1, \dots, r_j^n$ are t-consistent. In particular, this means that for each

$j' \in [2t+2, n]$, the vector of the j'-th elements of $\left(\mathbf{M}^{\mathsf{c}([2t+2,n])}_{\mathsf{r}([t+2,2t+1])}\right)^{-1} \cdot (r^k_{t+2}, \ldots, r^k_{2t+1})^\intercal$ across $k \in [n]$ are t-consistent. Similarly, for $l \in [2t+1]$, $s^1_l, \ldots, s^n_l$ are generated by honest parties and thus are t-consistent. Therefore, for each $j' \in [2t+2, n]$, the vector of the j'-th elements of $\left(\mathbf{M}^{\mathsf{c}([2t+2,n])}_{\mathsf{r}([t+2,2t+1])}\right)^{-1} \cdot \mathbf{M}^{\mathsf{c}([2t+1])}_{\mathsf{r}([t+2,2t+1])} \cdot (s^k_1 \ldots s^k_{2t+1})^\intercal$ across $k \in [n]$ are t-consistent. Putting together the observations above, we have that for each $j' \in [2t+2, n]$, the vector of the j'-th elements of $(s^k_{2t+2} \ldots s^k_n)^\intercal$ across $k \in [n]$ are t-consistent.

Finally, since for $k \in [n]$:

$$(r^k_1, \ldots, r^k_{t+1})^\intercal \leftarrow \mathbf{M}^{\mathsf{c}([2t+2,n])}_{\mathsf{r}([t+1])} \cdot (s^k_{2t+2} \ldots s^k_n)^\intercal + \mathbf{M}^{\mathsf{c}([2t+1])}_{\mathsf{r}([t+1])} \cdot (s^k_1 \ldots s^k_{2t+1})^\intercal,$$

we see that for $j \in [t+1]$, $(r^1_j, \ldots, r^n_j)$ are t-consistent. Thus, every $\left[e_j\right]^{C_{i+1}}_{t'} = [0]^{C_{i+1}}_0$, as required.

In conclusion, we have proved that the real and ideal worlds are distributed identically and thus $\Pi_{\mathsf{rand\text{-}2/3\text{-}hm}}$ UC-realizes $\mathcal{F}_{\mathsf{rand\text{-}2/3\text{-}hm}}$.

$\square$

Input and Output Functionality $\mathcal{F}_{\mathsf{input}}$ below models how clients provide inputs to the computation, and it is instantiated by Protocol Π_{input}, which is presented right after. The instantiation is quite standard: clients learn a random value that they can use to mask their input, sending this masked value to the parties in the committee that initiates the computation, who can unmask this element using secret-sharing. Here, we assume the client committee C_{clnt} is the first *two* committees, or in other words, clients have fluidity 2. This can be easily removed by placing one committee before C_{clnt} which send the random sharings to C_{clnt}, but this would require us to use our resharing protocol, which we present later in Section 3.5.2.2.

Figure 3.26: Functionality $\mathcal{F}_{\text{input}}$

1. Let x be the input associated with the input gate belonging to the client.
2. If the client is corrupted:
   (a) $\mathcal{F}_{\text{input}}$ first receives the sharing $[x]_{2t}^{C_{\text{clnt}}}$ or (abort, A) for $A \subseteq \mathcal{H}_{C_{\text{clnts}}}$ from the adversary.
   (b) In the former case $\mathcal{F}_{\text{input}}$ forwards to the honest parties their shares.
   (c) In the latter case, $\mathcal{F}_{\text{input}}$ outputs abort to the honest clients A of C_{clnt}.
3. If the client is honest:
   (a) $\mathcal{F}_{\text{input}}$ first receives x from the client, then it receives from the adversary a set of shares $\{x^j\}_{j \in \mathcal{T}_{C_{\text{clnts}}}}$ or abort.
   (b) In the former case, $\mathcal{F}_{\text{input}}$ then samples a sharing $[x]_{2t}^{C_{\text{clnt}}}$ based on the t shares x^j from the adversary, the value x, and t other randomly sampled shares, then sends to the honest parties their shares.
   (c) In the latter case, $\mathcal{F}_{\text{input}}$ outputs abort to the honest clients.

Figure 3.27: Protocol Π_{input}

Usage: A client in the client set C_{clnt} distributes a sharing $[x]_{2t}^{C_{\text{clnt}}}$ of their circuit input x to the other clients.

1. The clients P_j of C_{clnt} first invoke $\mathcal{F}_{\text{rand-2/3-hm}}$ to get random sharing $[r]_t^{C_{\text{clnt}}}$ (where C_{clnt} acts as all committees used in $\mathcal{F}_{\text{rand-2/3-hm}}$).
2. The clients then open $[r]_t^{C_{\text{clnt}}}$ to the input client, who then (if $[r]_t^{C_{\text{clnt}}}$ is a correct degree-t sharing) computes $x + r$, samples $[x+r]_{2t}^{C_{\text{clnt}}}$ and distributes to all other clients in C_{clnt} their shares.
3. Each party in C_{clnt} then computes and outputs their sharing of x as $[x]_{2t}^{C_{\text{clnt}}} = [x+r]_{2t}^{C_{\text{clnt}}} - [r]_t^{C_{\text{clnt}}}$.

Lemma 3.20. Π_{input} *UC-realizes* $\mathcal{F}_{\mathsf{input}}$ *in the* $\mathcal{F}_{\mathsf{rand\text{-}2/3\text{-}hm}}$*-hybrid model.*

Proof. Assume w.l.o.g., that the first $2t+1$ parties are honest. First, we define the simulator $\mathcal{S}$:

1. $\mathcal{S}$ first emulates $\mathcal{F}_{\mathsf{rand\text{-}hm}}$. That is, it receives from the adversary shares $r^{2t+2}, \ldots, r^n$ and sharing $[e]_{t'}^{C_{\mathsf{clnt}}}$, then samples r uniformly at random, generates random sharing $[r]_t^{C_{\mathsf{clnt}}}$ such that each k-th share is r^k, aborts if $[e]_{t'}^{C_{\mathsf{clnt}}} \neq [0]_0^{C_{\mathsf{clnt}}}$ and finally asks the adversary whether to continue. If the adversary responds with abort, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{input}}$.

2. Then, if the input client is corrupt:

   (a) $\mathcal{S}$ sends the honest clients' shares of $[r]_t^{C_{\mathsf{clnt}}}$ to the adversary.

   (b) If the client responds with abort to honest clients $A \in \mathcal{H}_{C_{\mathsf{clnts}}}$, $\mathcal{S}$ forwards it (abort, A) to $\mathcal{F}_{\mathsf{input}}$.

   (c) Otherwise, the client responds with the honest parties' shares of $[x+r]_{2t}^{C_{\mathsf{clnt}}}$ ($2t+1$ of them), from which $\mathcal{S}$ reconstructs the whole sharing and then forwards the sharing of $[x]_{2t}^{C_{\mathsf{clnt}}} = [x+r]_{2t}^{C_{\mathsf{clnt}}} - [r]_t^{C_{\mathsf{clnt}}}$ to $\mathcal{F}_{\mathsf{input}}$.

3. Otherwise, if the input client is honest:

   (a) $\mathcal{S}$ receives from the adversary the corrupt parties' shares of $[r]_t^{C_{\mathsf{clnt}}}$.

   (b) If any of the shares it receives are not equal to r^k received above, $\mathcal{S}$ sends abort to $\mathcal{F}_{\mathsf{input}}$.

   (c) Otherwise, $\mathcal{S}$ sends random $s^{2t+2}, \ldots, s^n$ to the adversary and on behalf of the corrupt parties, computes their shares $(s^{2t+2} - r^{2t+2}, \ldots, s^n - r^n)$ and forwards them to $\mathcal{F}_{\mathsf{input}}$.

Now we argue that the real world and ideal world are distributed identically to the adversary. It is clear that $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{rand\text{-}hm}}$ exactly. If the input client is corrupt, then in both the real world and ideal world, the adversary first receives from the honest clients their shares of $[r]_t^{C_{\mathsf{clnt}}}$ (in the latter case, via the simulator $\mathcal{S}$). Then, based on the adversary's sharing $[x+r]_{2t}^{C_{\mathsf{clnt}}}$, the honest clients in both worlds output $[x+r]_{2t}^{C_{\mathsf{clnt}}} - [r]_t^{C_{\mathsf{clnt}}}$. Therefore, the two worlds are distributed identically in this case.

If the client is honest, then in the real world, the client first checks if the shares of $[r]_t^{C_{\mathsf{clnt}}}$ it receives are t-consistent. In the ideal world, since the only corrupt party shares that are t-consistent with the honest parties' shares are those that $\mathcal{S}$ received from the adversary originally, $\mathcal{S}$ properly aborts if it does not receive these shares from the adversary. Then, in the real world, the corrupt parties receive from the honest client their shares of fresh sharing $[x+r]_{2t}^{C_{\mathsf{clnt}}}$. Since this is a fresh sharing, by the properties of Shamir secret sharing, the corrupt parties' shares are uniformly random, which is what $\mathcal{S}$ sends to the adversary in the ideal world. Finally, based on the sharing $[x+r]_{2t}^{C_{\mathsf{clnt}}}$, the honest clients in the real world output $[x+r]_{2t}^{C_{\mathsf{clnt}}} - [r]_t^{C_{\mathsf{clnt}}}$. Note that the adversary only has t shares of $[x+r]_{2t}^{C_{\mathsf{clnt}}}$, and thus, there are at least t random degrees of freedom left in it. Therefore, since $\mathcal{S}$ computes the corrupt parties' shares of $[x+r]_{2t}^{C_{\mathsf{clnt}}} - [r]_t^{C_{\mathsf{clnt}}}$ exactly as in the real world, as $(s^{2t+2} - r^{2t+2}, \ldots, s^n - r^n)$, then $\mathcal{F}_{\mathsf{input}}$ computes $[x]_{2t}^{C_{\mathsf{clnt}}}$ based on these t shares, the value x, and t other randomly sampled shares; the shares of $[x]_{2t}^{C_{\mathsf{clnt}}}$ output by honest clients in the ideal world are distributed identically to those of the real world.

Therefore, the real and ideal worlds are distributed identically. $\square$

Finally, functionality $\mathcal{F}_{\mathsf{output}}$ models how the clients get output, and its instantiation, Protocol Π_{output}, is given right after. The protocol is quite simple: the parties from the committee holding shares of the output send their shares to the receiving client, who performs error detection to reconstruct the correct output (or abort).

Figure 3.28: Functionality $\mathcal{F}_{\mathsf{output}}$

1. Let $[z]_{2t}^{C_\ell}$ be the sharing associated with the output gate which belongs to the client, held by the final committee C_ℓ.

2. $\mathcal{F}_{\mathsf{output}}$ first receives the shares of $[z]_{2t}^{C_\ell}$ from the honest parties $\mathcal{H}_{C_\ell}$ and uses them to reconstruct all of $[z]_{2t}^{C_\ell}$.

3. Depending on whether the client is honest or not, there are two cases:

   (a) If the client is corrupted, $\mathcal{F}_{\mathsf{output}}$ sends the whole sharing $[z]_{2t}^{C_\ell}$ to the adversary. If the adversary replies (abort, A) for $A \subseteq \mathcal{H}_{C_{\mathsf{clnts}}}$, $\mathcal{F}_{\mathsf{output}}$ sends abort to all honest clients A of C_{clnt}.

   (b) If the client is honest, $\mathcal{F}_{\mathsf{output}}$ sends just the corrupt parties' shares of $[z]_{2t}^{C_\ell}$ to the adversary. Then, $\mathcal{F}_{\mathsf{output}}$ asks the adversary whether it should continue. If the adversary replies abort, $\mathcal{F}_{\mathsf{output}}$ sends abort to all honest parties. Otherwise, $\mathcal{F}_{\mathsf{output}}$ sends z to the client.

Figure 3.29: Protocol Π_{output}

Usage: The last committee C_ℓ reconstructs to a client the sharing $[z]_{2t}^{C_\ell}$ of their output z.

1. The last committee C_ℓ simply opens output $[z]_{2t}^{C_\ell}$ to the client, P_j of C_{clnt}.

2. The client P_j then checks if $[z]_{2t}^{C_\ell}$ is a correct degree-$2t$ sharing. If so, it outputs z; otherwise, it sends abort to all other clients in C_{clnt}.

Lemma 3.21. *Π_{output} UC-realizes $\mathcal{F}_{\mathsf{output}}$.*

Proof. First, we define the simulator $\mathcal{S}$:

1. If the client is corrupted:

   (a) $\mathcal{S}$ receives from $\mathcal{F}_{\mathsf{output}}$ the whole sharing $[z]_{2t}^{C_\ell}$ and forwards to the adversary the honest parties' shares.

   (b) Finally, if $\mathcal{S}$ receives on behalf of the honest clients $A \subseteq \mathcal{H}_{C_{\mathsf{clnts}}}$, abort, from the adversary, $\mathcal{S}$ sends (abort, A) to $\mathcal{F}_{\mathsf{output}}$.

2. If the client is honest:

(a) $\mathcal{S}$ first receives from $\mathcal{F}_{\mathsf{output}}$ the corrupt parties' shares $\{z_j\}_{j\in\mathcal{T}_{C_\ell}}$ of $[z]_{2t}^{C_\ell}$.

(b) Then, $\mathcal{S}$ receives on behalf of the honest client $\{z'_j\}_{j\in\mathcal{T}_{C_\ell}}$ from the adversary.

(c) If any $z'_j \neq z_j$, then $\mathcal{S}$ sends abort to the corrupt clients of $\mathcal{T}_{C_{\mathsf{clnts}}}$ and then also to $\mathcal{F}_{\mathsf{output}}$.

Now we show that the real and ideal world are identically distributed to the adversary. If the client is corrupted, it is clear that in both worlds, the adversary receives the honest parties' shares of $[z]_{2t}^{C_\ell}$. Thus, the two worlds are identically distributed in this case.

If the client is honest, in the real world they receive all of the committee C_ℓ parties' shares of $[z]_{2t}^{C_\ell}$ and if the shares are not $2t$-consistent, then the client sends abort to all other clients; otherwise, they output z. In the ideal world $\mathcal{S}$ receives from $\mathcal{F}_{\mathsf{output}}$ the shares of the corrupt parties of C_ℓ that are (the only shares that are) $2t$-consistent with the honest parties. Therefore, since $\mathcal{S}$ aborts if and only if any shares it receives from the adversary are different from those received from $\mathcal{F}_{\mathsf{output}}$, the simulation is perfect in this case. □

3.5.2.2 Robust Linear-Overhead Resharing

A central building block needed for Fluid MPC protocols consists of transferring secret-shared values from one committee to the next. This is needed at least to transfer the "state" of the computation itself, but in our case, as in [Bienstock et al. 2023a], it is also used to transfer other information such as multiplication triples, in order to get efficient multiplication. In this section we present efficient resharing protocols for the case of perfect security with abort. In more detail, consider a committee C_i that holds sharings $[x]_{2t}^{C_i}$. Ideally, we would design a protocol such that committee C_{i+1} receives sharings $[x]_{2t}^{C_{i+1}}$. Unfortunately, one can easily hint why such primitive is hard to instantiate with *linear* communication: there have to be at least t honest parties in C_{i+1} who derive their shares from more than t messages from C_i, since otherwise, the adversary corrupting t partiesin C_i could learn these t receivers' messages and hence their shares. Instead, we design a protocol with *linear* communication that lets committee C_{i+2} obtain shares $[x]_{2t}^{C_i}$, instead of C_{i+1}. This is sufficient for our main protocol.

Our protocol operates in *batches*, resharing a group of $t + 1 = \Omega(n^2)$ sharings with communication $O(n^2)$, which is linear amortized. The description of our protocol is divided into two steps. First, the parties in C_i reshare the $t + 1$ secrets into an intermediate *packed* version of them towards committee C_{i+1}. This is described in Section 3.5.2.2 below. Next, parties in C_{i+1} somehow "unpack" these sharings so that the parties in C_{i+2} obtain standard Shamir sharings of the original batch. This second part appears in Section 3.5.2.2.

Robust Resharing from Standard to Packed Consider $t + 1$ sharings $([x_1]_{2t}^{C_i}, \ldots, [x_{t+1}]_{2t}^{C_i})$ held by a committee C_i. We first present a protocol in which the parties in C_{i+1} can obtain packed secret-sharings $[\mathbf{x}]_{2t}^{C_{i+1}}$, where $\mathbf{x} = (x_1, \ldots, x_{t+1})$. The formal functionality is described below as $\mathcal{F}_{\text{robust-packed-reshare}}$. Notice that there are some technicalities involved in the definition of the functionality. First, corrupt parties in C_i can cheat in this protocol and cause the parties in C_{i+1} to obtain incorrect sharings $[\mathbf{x} + \mathbf{e}]_{2t}^{C_{i+1}}$, where $\mathbf{e}$ is some error vector chosen by the adversary. Our protocol guarantees that such error will be catched by the parties in C_{i+3}, so the functionality models this fact by sending abort to the parties in this committee. In addition, the adversary may cause some of the parties in C_{i+3} to abort (selectively), and the functionality also accounts for this.

Figure 3.30: Functionality $\mathcal{F}_{\text{robust-packed-reshare}}$

1. Let $([x_1]_{2t}^{C_i}, \ldots, [x_{t+1}]_{2t}^{C_i})$ be sharings held by the parties of C_i, corresponding to the vector of values $\mathbf{x} = (x_1, \ldots, x_{t+1})$.

2. $\mathcal{F}_{\text{robust-packed-reshare}}$ first receives from the honest parties of C_i their shares of $([x_1]_{2t}^{C_i}, \ldots, [x_{t+1}]_{2t}^{C_i})$ (at least $2t + 1$ for each of them).

3. $\mathcal{F}_{\text{robust-packed-reshare}}$ then reconstructs all of $([x_1]_{2t}^{C_i}, \ldots, [x_{t+1}]_{2t}^{C_i})$ and sends the shares of corrupted parties to the adversary.

4. $\mathcal{F}_{\text{robust-packed-reshare}}$ then receives from the adversary a set of shares $\{\mathbf{x}^k\}_{k \in \mathcal{T}_{C_{i+1}}}$, and error

vector $\mathbf{e}$.

5. Next, $\mathcal{F}_{\text{robust-packed-reshare}}$ computes the sharing $[\mathbf{x}+\mathbf{e}]_{2t}^{C_{i+1}}$ based on the t shares $\mathbf{x}^k$ from the adversary and the vector $\mathbf{x}+\mathbf{e}$, then sends to the honest parties their shares.

6. Finally, if $\mathbf{e} \neq (0, \ldots, 0)$, $\mathcal{F}_{\text{robust-packed-reshare}}$ sends abort to the honest parties of C_{i+3}. Otherwise, $\mathcal{F}_{\text{robust-packed-reshare}}$ asks the adversary whether to continue, and if the adversary replies (abort, A) for $A \subseteq \mathcal{H}_{C_{i+3}}$, then $\mathcal{F}_{\text{robust-packed-reshare}}$ sends abort to the honest parties A of C_{i+3}.

Our protocol is conceptually simple: the parties in C_i each collect their shares of the batch of secrets, and distributes packed sharings of these. The receiving parties in C_{i+1} can then take an appropriate linear combination of these sharings (using Lagrange coefficients) to derive packed sharings of the underlying vector of secrets. To prevent a corrupt party from sending shares of an incorrect vector, we use the parity-check matrix defined in Section 3.1.3: we let $\mathbf{H}$ denote the $(t+1)\times n$ matrix such that $\mathbf{H}\cdot(x^1, \ldots, x^n)^\intercal = (0, \ldots, 0)^\intercal$ if and only if $(x^1, \ldots, x^n)$ are $2t$-consistent. Via $\mathbf{H}$, checking if a group of values is $2t$-consistent reduces to applying a linear combination on these and checking that the result is zero. The parties in C_{i+1} can apply such combination to their received packed sharings, which can be checked by the parties in a future committee. Instead of reconstructing this linear combination to the parties in C_{i+2}, which would be too expensive, the parties perform the linear reconstruction from [Damgård and Nielsen 2007] that reconstructs these sharings to the parties in committee C_{i+3}, who check that the results are zero.

Our protocol is presented formally as Protocol $\Pi_{\text{robust-packed-reshare}}$ below. We note here that $\Pi_{\text{robust-packed-reshare}}$ can successfully be used on input sharings $([x_1]_t^{C_i}, \ldots, [x_{t+1}]_t^{C_i})$ of degree-t instead of degree-$2t$. Indeed, the only property that is required of the original sharings $[x_\alpha]_t^{C_i}$ for $\alpha \in [t+1]$ is that their shares lie on a polynomial of degree *at most 2t*, which is of course true. We will use this fact in our main Fluid MPC protocol.

Figure 3.31: Protocol $\Pi_{\text{robust-packed-reshare}}$

Usage: Committee C_i holds standard sharings $([x_1]_{2t}^{C_i}, \ldots, [x_{t+1}]_{2t}^{C_i})$ corresponding to the vector of values $\mathbf{x} = (x_1, \ldots, x_{t+1})$ and committee C_{i+1} outputs a single packed sharing $[\mathbf{x}]_{2t}^{C_{i+1}}$, using committees C_{i+2} and C_{i+3} to ensure its correctness.

1. Every party P_j in committee C_i distributes degree-$2t$ packed sharings $\left[\mathbf{x}^j_{[1,t+1]}\right]_{2t}^{C_{i+1}}$ to committee C_{i+1} of its shares $\mathbf{x}^j_{[1,t+1]} = (x_1^j, \ldots, x_{t+1}^j)$ of the sharings $([x_1]_{2t}^{C_i}, \ldots, [x_{t+1}]_{2t}^{C_i})$ of the values in $\mathbf{x}$.

2. While the Verification phase (below) executes, the parties of committee C_{i+1} compute and output fresh packed shares $[\mathbf{x}]_{2t}^{C_{i+1}} = \sum_{j=1}^{2t+1} L_j(0) \cdot \left[\mathbf{x}^j_{[1,t+1]}\right]_{2t}^{C_{i+1}}$, where $L_j(0)$ are Lagrange interpolation coefficients.

Verification Phase:

1. The parties P_k of committee C_{i+1} apply the parity check matrix $\mathbf{H}$ to get $([\mathbf{y}_1]_{2t}^{C_{i+1}}, \ldots, [\mathbf{y}_{n-2t-1}]_{2t}^{C_{i+1}})^\intercal \leftarrow \mathbf{H} \cdot (\left[\mathbf{x}^1_{[1,t+1]}\right]_{2t}^{C_{i+1}}, \ldots, \left[\mathbf{x}^n_{[1,t+1]}\right]_{2t}^{C_{i+1}})^\intercal$.

2. Next, they apply super-invertible matrix $\mathbf{M}$ to get $([\mathbf{z}_1]_{2t}^{C_{i+1}}, \ldots, [\mathbf{z}_n]_{2t}^{C_{i+1}})^\intercal \leftarrow \mathbf{M} \cdot ([\mathbf{y}_1]_{2t}^{C_{i+1}}, \ldots, [\mathbf{y}_{n-2t-1}]_{2t}^{C_{i+1}})^\intercal$, and open $[\mathbf{z}_l]_{2t}^{C_{i+1}}$ to party P_l of committee C_{i+2}.

3. Finally, each party P_l of committee C_{i+2} checks that the shares of $[\mathbf{z}_l]_{2t}^{C_{i+1}}$ are $2t$-consistent and that they correspond to $\mathbf{z}_l = 0^\ell$.

4. If either of the checks fail, they send abort to the parties of committee C_{i+3}.

The communication complexity of $\Pi_{\text{robust-packed-reshare}}$ is $n \cdot n = O(n^2)$ sharings in step 1, plus $n \cdot n = O(n^2)$ sharings from second step in the verification, for a total of $O(n^2/(t+1)) = O(n)$ per value being reshared.

Lemma 3.22. $\Pi_{\text{robust-packed-reshare}}$ *UC-realizes* $\mathcal{F}_{\text{robust-packed-reshare}}$.

Proof. First, we define the simulator $\mathcal{S}$:

1. $\mathcal{S}$ first receives from $\mathcal{F}_{\text{robust-packed-reshare}}$ the shares of $([x_1]_{2t}^{C_i}, \ldots, [x_{t+1}]_{2t}^{C_i})$ of the corrupted parties: $(x_1^j, \ldots, x_{t+1}^j)_{j \in \mathcal{T}_{C_i}}$

2. For each $j' \in \mathcal{H}_{C_i}$ and $k' \in \mathcal{T}_{C_{i+1}}$, $\mathcal{S}$ samples random $x_{j'k'}$ and sends it to the adversary, to emulate $P_{k'}$'s share of $P_{j'}$'s packed sharing $\left[\mathbf{x}_{[1,t+1]}^{j'}\right]_{2t}^{C_{i+1}}$ in Step 1 of $\Pi_{\text{robust-packed-reshare}}$.

3. $\mathcal{S}$ then receives from the adversary for each $j \in \mathcal{T}_{C_i}$ and $k \in \mathcal{H}_{C_{i+1}}$, value x_{jk}, corresponding to P_k's share of P_j's packed sharing $\left[\hat{\mathbf{x}}_{[1,t+1]}^{j}\right]_{2t}^{C_{i+1}}$. For each j, $\mathcal{S}$ uses the $2t+1$ values x_{jk} to compute the sharing $\left[\hat{\mathbf{x}}_{[1,t+1]}^{j}\right]_{2t}^{C_{i+1}} = (x_{j1}, \ldots, x_{jn})$, and the underlying block of secrets $\hat{\mathbf{x}}_{[1,t+1]}^{j} = (\hat{x}_1^j, \ldots, \hat{x}_{t+1}^j)$.

4. Next, $\mathcal{S}$ computes for each $j \in \mathcal{T}_{C_i}$, their error vector

$$\mathbf{e}_j = (e_{j,1}, \ldots, e_{j,t+1}) \leftarrow (\hat{x}_1^j - x_1^j, \ldots, \hat{x}_{t+1}^j - x_{t+1}^j),$$

and then the overall error vector

$$\mathbf{e} = \sum_{j \in \mathcal{T}_{C_i} \cap [2t+1]} L_j(0) \cdot \mathbf{e}_j.$$

5. $\mathcal{S}$ then computes for $k' \in \mathcal{T}_{C_{i+1}}$, $\mathbf{x}^{k'} = \sum_{j=1}^{2t+1} L_j(0) \cdot x_{jk'}$ and sends these values, along with $\mathbf{e}$ to $\mathcal{F}_{\text{robust-packed-reshare}}$.

6. Next, $\mathcal{S}$ computes for each $\alpha \in [t+2]$, the matrix-vector product

$$(y_{1,\alpha}, \ldots, y_{n-2t-1,\alpha})^\top \leftarrow \mathbf{H} \cdot (e_{1,\alpha}, \ldots, e_{n,\alpha})^\top,$$

where $e_{j,\alpha} = 0$ for all $j \in \mathcal{H}_{C_i}$, $\alpha \in [t+1]$, and defines the vector $\mathbf{y}_\mu = (y_{\mu,1}, \ldots, y_{\mu,t+1})$ for $\mu \in [n-2t-1]$.

7. $\mathcal{S}$ also computes for each $k' \in \mathcal{T}_{C_{i+1}}$:

$$(y_1^{k'}, \ldots, y_{n-2t-1}^{k'})^\intercal \leftarrow \mathbf{H} \cdot (x_{1k'}, \ldots, x_{nk'})^\intercal .$$

8. For each $\mu \in [n - 2t - 1]$, using the blocks $\mathbf{y}_\mu$ and the shares $y_\mu^{k'}$ of $k' \in \mathcal{T}_{C_{i+1}}$ (i.e., $2t + 1$ total degrees of freedom), $\mathcal{S}$ computes the sharing $\left[\mathbf{y}_\mu\right]_{2t}^{C_{i+1}}$.

9. Next, $\mathcal{S}$ computes $([\mathbf{z}_1]_{2t}^{C_{i+1}}, \ldots, [\mathbf{z}_n]_{2t}^{C_{i+1}}) \leftarrow \mathbf{M} \cdot ([\mathbf{y}_1]_{2t}^{C_{i+1}}, \ldots, [\mathbf{y}_{n-2t-1}]_{2t}^{C_{i+1}})$, and sends the honest parties' shares of $[\mathbf{z}_l]_{2t}^{C_{i+1}}$ to each corrupted party P_l of committee C_{i+2}.

10. For each honest party P_l of committee C_{i+2}, $\mathcal{S}$ receives from the adversary: committee C_{i+1} corrupted parties' shares of $[\mathbf{z}_l]_{2t}^{C_{i+1}}$. If these shares do not match what $\mathcal{S}$ had already computed, or the underlying computed value $\mathbf{z}_l \neq (0, \ldots, 0)$, $\mathcal{S}$ sends abort on behalf of P_l to all corrupted parties of committee C_{i+3}.

11. Finally, if $\mathcal{S}$ sent any abort in the above step it sets $A = \mathcal{H}_{C_{i+3}}$; otherwise, it sets A to be those honest parties that receive abort from any corrupt party abort. Then, when $\mathcal{F}_{\text{robust-packed-reshare}}$ asks $\mathcal{S}$ whether to continue, it replies with abort.

Now we show that the real world and ideal world are identically distributed to the adversary. In Step 1 of $\Pi_{\text{robust-packed-reshare}}$, the adversary receives the t committee C_{i+1} corrupted parties' shares of the committee C_i honest parties' packed sharings $\left[\mathbf{x}_{[1,t+1]}^j\right]_{2t}^{C_{i+1}}$. Since these are degree-$2t$ packed sharings that share $t + 1$ underlying values, there are t remaining degrees of freedom in defining the underlying polynomial, and thus the shares that the adversary sees are uniformly random. Indeed, Step 2 of $\mathcal{S}$ is to send uniformly random shares to the adversary, corresponding to the packed sharings, and thus the view of the adversary is identical in the real and ideal worlds for this step.

In Step 2 of $\Pi_{\text{robust-packed-reshare}}$, the honest parties output their shares $[\mathbf{x}]_{2t}^{C_{i+1}} = \sum_{j=1}^{2t+1} L_j(0) \cdot \left[\mathbf{x}_{[1,t+1]}^j\right]_{2t}^{C_{i+1}}$ based on their own as well as possibly some corrupted sharings $\left[\mathbf{x}_{[1,t+1]}^j\right]_{2t}^{C_{i+1}}$. The honest parties' shares uniquely define a degree-$2t$ polynomial, where for $\alpha \in [t+1]$ the polynomial evaluated on $-\alpha$ gives

$$\sum_{k\in\mathcal{H}_{C_{i+1}}} L_k(-\alpha)\sum_{j=1}^{2t+1} L_j(0)\cdot x_{jk} = \sum_{j=1}^{2t+1} L_j(0)\sum_{k\in\mathcal{H}_{C_{i+1}}} L_k(-\alpha)x_{jk} = \sum_{j=1}^{2t+1} L_j(0)\hat{x}_\alpha^j$$

$$= \sum_{j\in\mathcal{H}_{C_{i+1}}\cap[2t+1]} L_j(0)x_\alpha^j + \sum_{j\in\mathcal{T}_{C_{i+1}}\cap[2t+1]} L_j(0)\cdot(x_\alpha^j + e_{j,\alpha}) = x_\alpha + \sum_{j\in\mathcal{T}_{C_{i+1}}\cap[2t+1]} L_j(0)\cdot e_{j,\alpha},$$

where $e_{j,\alpha}$ is some error injected by the adversary for corrupt party P_j. Similarly, for $k' \in \mathcal{T}_{C_{i+1}}$, the polynomial evaluated on k', i.e., $P_{k'}$'s share, gives

$$\sum_{j=1}^{2t+1} L_j(0)\sum_{k\in\mathcal{H}_{C_{i+1}}} L_k(k')x_{jk} = \sum_{j=1}^{2t+1} L_j(0)x_{jk'},$$

where $x_{jk'}$ is the corrupted party $P_{k'}$'s share of $\left[\mathbf{x}^j_{[1,t+1]}\right]_{2t}^{C_{i+1}}$. In the ideal world, $\mathcal{S}$ in Step 3 for $j \in \mathcal{T}_{C_i}$ reconstructs the entire packed sharings $\left[\mathbf{x}^j_{[1,t+1]}\right]_{2t}^{C_{i+1}}$. This allows $\mathcal{S}$ to reconstruct the corresponding corrupted party $P_{k'}$'s shares of each of these packed sharings, $x_{jk'}$, as well as the underlying $(\hat{x}_1^j, \dots, \hat{x}_{t+1}^j)$, which are supposed to be P_j's shares of the original $(t+1)$ standard sharings. The former allows it to, along with the corrupted party's shares of honest party's sharings $\left[\mathbf{x}^j_{[1,t+1]}\right]_{2t}^{C_{i+1}}$, compute the corrupted parties' shares of $[\mathbf{x}]_{2t}^{C_{i+1}}$, which correspond to exactly those in the real world, as computed above. The latter allows it to compute the error $\mathbf{e}_j$ of these shares, as compared to those received from $\mathcal{F}_{\text{robust-packed-reshare}}$ in Step 1, and from this, the overall error $\mathbf{e}$, which is exactly the error in the real world, as computed above. Finally, $\mathcal{S}$, gives to $\mathcal{F}_{\text{robust-packed-reshare}}$, the corrupted party's shares of $[\mathbf{x}]_{2t}^{C_{i+1}}$, and the error $\mathbf{e}$, from which $\mathcal{F}_{\text{robust-packed-reshare}}$ further computes the perturbed underlying vector $\mathbf{x}+\mathbf{e}$. This gives $\mathcal{F}_{\text{robust-packed-reshare}}$ $2t+1$ points on a degree-$2t$ polynomial, which are exactly the same in the real world, and based on this $\mathcal{F}_{\text{robust-packed-reshare}}$ reconstructs $[\mathbf{x}]_{2t}^{C_{i+1}}$ and gives honest parties their shares. Thus, the shares output by the honest parties in the real and ideal world are distributed identically.

In Step 2 of the **Verification Phase** of $\Pi_{\text{robust-packed-reshare}}$, the adversary receives on behalf of corrupt party P_l of committee C_{i+2}, the honest parties' shares of $[\mathbf{z}_l]_{2t}^{C_{i+1}}$. These shares come from them computing

$$\mathbf{M} \cdot \mathbf{H} \cdot \left(\left[\hat{\mathbf{x}}^1_{[1,t+1]}\right]_{2t}^{C_{i+1}}, \ldots, \left[\hat{\mathbf{x}}^n_{[1,t+1]}\right]_{2t}^{C_{i+1}} \right)^\intercal =$$

$$\mathbf{M} \cdot \mathbf{H} \cdot \left(\left(\left[\mathbf{x}^1_{[1,t+1]}\right]_{2t}^{C_{i+1}}, \ldots, \left[\mathbf{x}^n_{[1,t+1]}\right]_{2t}^{C_{i+1}} \right)^\intercal + \left([\mathbf{e}_1]_{2t}^{C_{i+1}}, \ldots, [\mathbf{e}_n]_{2t}^{C_{i+1}} \right)^\intercal \right),[11]$$

where $\left[\mathbf{e}_j\right]_{2t}^{C_{i+1}} = 0$ for $j \in \mathcal{H}_{C_i}$,

$$= ([\mathbf{0}]_{2t}^{C_{i+1}}, \ldots, [\mathbf{0}]_{2t}^{C_{i+1}})^\intercal + \mathbf{M} \cdot \mathbf{H} \cdot \left([\mathbf{e}_1]_{2t}^{C_{i+1}}, \ldots, [\mathbf{e}_n]_{2t}^{C_{i+1}} \right)^\intercal,$$

where the identity of the first term is due to the fact that the underlying shares $x^1_\alpha, \ldots, x^n_\alpha$ that are packed into the α-th slot of the respective packed sharings $\left[\mathbf{x}^j_{[1,t+1]}\right]_{2t}^{C_{i+1}}$, for $\alpha \in [t+1]$ indeed correspond to valid points of a polynomial of degree $\leq 2t$, and therefore applying H to them results in zeroes in the α-th slot of each element of the output vector.

In the ideal world, $\mathcal{S}$ computes $\mathbf{H} \cdot (\mathbf{e}_1, \ldots, \mathbf{e}_n)^\intercal$, where $\mathbf{e}_j = \mathbf{0}$ for $j \in \mathcal{H}_{C_i}$, and the corrupted parties' shares of $([\mathbf{y}_1]_{2t}^{C_{i+1}}, \ldots, [\mathbf{y}_{n-2t-1}]_{2t}^{C_{i+1}})^\intercal \leftarrow \mathbf{H} \cdot \left(\left[\hat{\mathbf{x}}^1_{[1,t+1]}\right]_{2t}^{C_{i+1}}, \ldots, \left[\hat{\mathbf{x}}^n_{[1,t+1]}\right]_{2t}^{C_{i+1}} \right)^\intercal$. This gives $\mathcal{S}$: $2t+1$ evaluations of the polynomials underlying $([\mathbf{y}_1]_{2t}^{C_{i+1}}, \ldots, [\mathbf{y}_{n-2t-1}]_{2t}^{C_{i+1}})$; from which it can reconstruct the whole sharings. It can then compute

$$([\mathbf{z}_1]_{2t}^{C_{i+1}}, \ldots, [\mathbf{z}_n]_{2t}^{C_{i+1}})^\intercal \leftarrow \mathbf{M} \cdot ([\mathbf{y}_1]_{2t}^{C_{i+1}}, \ldots, [\mathbf{y}_{n-2t-1}]_{2t}^{C_{i+1}})^\intercal,$$

from which it can send the corrupted parties P_l all of the honest parties' shares of $[\mathbf{z}_l]_{2t}^{C_{i+1}}$. Thus, the real world and ideal world are identically distributed at this step.

[11] Such decomposition of the maliciously shared $\left[\hat{\mathbf{x}}^j_{[1,t+1]}\right]_{2t}^{C_{i+1}}$ is always possible since we can define some canonical packed sharing $\left[\mathbf{x}^j_{[1,t+1]}\right]_{2t}^{C_{i+1}} := (x_{j1}, \ldots, x_{jn})$, and $\left[\mathbf{e}_j\right]_{2t}^{C_{i+1}}$ as the rest.

In step 2 of the **Verification Phase** of $\Pi_{\text{robust-packed-reshare}}$, if the corrupted parties' shares of $[\mathbf{z}_l]_{2t}^{C_{i+1}}$ are not consistent with those of the honest parties, then we know from the error-detection properties of packed sharings that honest party P_l will send abort to the parties of committee C_{i+3}, and similarly if $\mathbf{z}_l \neq 0^{t+1}$. In the ideal world, $\mathcal{S}$ in Step 10, first checks if the corrupted parties' shares of $[\mathbf{z}_l]_{2t}^{C_{i+1}}$, for honest party P_l of C_{i+2}, match that which it had already computed (i.e., are consistent with those of honest parties), and then also if $\mathbf{z}_l = (0, \dots, 0)$. If either of these checks fail, $\mathcal{S}$ sends abort on behalf of P_l to all corrupted parties of committee C_{i+3}. Thus, the real world and ideal world are identically distributed at this step.

Finally, we need to show that if $\mathbf{e} \neq 0^{t+1}$, then the honest parties abort in the real world (as they are forced to in the ideal world). We do so by showing the contrapositive: i.e., if the honest parties in the real world do not abort, then $\mathbf{e} = 0$. If the honest parties do not abort, then it must be that for each of the $2t + 1$ honest parties P_l in committee C_{i+2}, $[\mathbf{z}_l]_{2t}^{C_{i+1}}$ are $2t$-consistent and correspond to $\mathbf{z}_l = 0^{t+1}$. By the error-correction properties of super-invertible matrix $\mathbf{M}$, this must also mean that $\mathbf{y}_1 = \dots = \mathbf{y}_{n-2t-1} = 0^{t+1}$ (because any $n - 2t - 1 \leq 2t + 1$ of the honest codeword symbols can be multiplied by the inverse of the corresponding submatrix of M to get back the original message, which therefore must be all zeroes). As written above, we have that $([\mathbf{y}_1]_{2t}^{C_{i+1}}, \dots, [\mathbf{y}_{n-2t-1}]_{2t}^{C_{i+1}})^\intercal =$

$$\mathbf{H} \cdot \left(\left(\left[\mathbf{x}^1_{[1,t+1]} \right]_{2t}^{C_{i+1}}, \dots, \left[\mathbf{x}^n_{[1,t+1]} \right]_{2t}^{C_{i+1}} \right)^\intercal + \left([\mathbf{e}_1]_{2t}^{C_{i+1}}, \dots, [\mathbf{e}_n]_{2t}^{C_{i+1}} \right)^\intercal \right),$$

where $\left[\mathbf{e}_j\right]_{2t}^{C_{i+1}} = 0$ for $j \in \mathcal{H}_{C_i}$,

$$= ([\mathbf{0}]_{2t}^{C_{i+1}}, \dots, [\mathbf{0}]_{2t}^{C_{i+1}})^\intercal + \mathbf{H} \cdot \left([\mathbf{e}_1]_{2t}^{C_{i+1}}, \dots, [\mathbf{e}_n]_{2t}^{C_{i+1}} \right)^\intercal .$$

Assume w.l.o.g., now that the first $2t + 1$ parties are honest, so that the above is equal to:

$$= ([\mathbf{0}]_{2t}^{C_{i+1}}, \dots, [\mathbf{0}]_{2t}^{C_{i+1}})^\intercal + \mathbf{H} \cdot \left(0, \dots, 0, [\mathbf{e}_{2t+2}]_{2t}^{C_{i+1}}, \dots, [\mathbf{e}_n]_{2t}^{C_{i+1}} \right)^\intercal .$$

Let us denote the matrix of shares of the sharings $([\mathbf{y}_1]_{2t}^{C_{i+1}}, \ldots, [\mathbf{y}_{n-2t-1}]_{2t}^{C_{i+1}})$ as $\mathbf{Y}$ (each column is the set of shares held by each party). Then,

$$\mathbf{Y} = \begin{pmatrix} \mathbf{y}_1^1 & \cdots & \mathbf{y}_1^n \\ \vdots & \ddots & \vdots \\ \mathbf{y}_{n-2t-1}^1 & \cdots & \mathbf{y}_{n-2t-1}^n \end{pmatrix} = \begin{pmatrix} \mathbf{0}_1^1 & \cdots & \mathbf{0}_1^n \\ \vdots & \ddots & \vdots \\ \mathbf{0}_{n-2t-1}^1 & \cdots & \mathbf{0}_{n-2t-1}^n \end{pmatrix} + \mathbf{H} \cdot \begin{pmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \\ \mathbf{e}_{2t+2}^1 & \cdots & \mathbf{e}_{2t+2}^n \\ \vdots & \ddots & \vdots \\ \mathbf{e}_n^1 & \cdots & \mathbf{e}_n^n \end{pmatrix}.$$

Testing if a vector $\mathbf{x} \in \mathbb{F}^{2t+1}$ corresponds to a degree-$2t$ packed sharing of $\mathbf{0} \in \mathbb{F}^{t+1}$ can be done by interpolating a polynomial f of degree $\leq 2t$ from the $2t + 1$ entries, and checking that $f(-\alpha) = 0$ for each $\alpha \in [t + 1]$. This can be directly expressed as a matrix product $\mathbf{0} = \mathbf{G} \cdot \mathbf{x}$, where $\mathbf{G}$ is a $(n-2t) \times (2t+1)$ matrix (since $n - 2t = t + 1$). Since each $\mathbf{y}_j$ is equal to $\mathbf{0} \in \mathbb{F}^{t+1}$, the honest entries of each row of Y, multiplied by $\mathbf{G}^\intercal$, gives zero, or in other words:

$$0 = \begin{pmatrix} \mathbf{y}_1^1 & \cdots & \mathbf{y}_1^n \\ \vdots & \ddots & \vdots \\ \mathbf{y}_{n-2t-1}^1 & \cdots & \mathbf{y}_{n-2t-1}^n \end{pmatrix} \cdot \left(\frac{\mathbf{G}^\intercal}{0}\right) = \overbrace{\begin{pmatrix} \mathbf{0}_1^1 & \cdots & \mathbf{0}_1^n \\ \vdots & \ddots & \vdots \\ \mathbf{0}_{n-2t-1}^1 & \cdots & \mathbf{0}_{n-2t-1}^n \end{pmatrix} \cdot \left(\frac{\mathbf{G}^\intercal}{0}\right)}^{0}$$
$$+ \mathbf{G} \cdot \begin{pmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \\ \mathbf{e}_{2t+2}^1 & \cdots & \mathbf{e}_{2t+2}^n \\ \vdots & \ddots & \vdots \\ \mathbf{e}_n^1 & \cdots & \mathbf{e}_n^n \end{pmatrix} \cdot \left(\frac{\mathbf{G}^\intercal}{0}\right)$$

Denoting

$$\mathbf{E} = \begin{pmatrix} \mathbf{e}_{2t+2}^1 & \cdots & \mathbf{e}_{2t+2}^{2t+1} \\ \vdots & \ddots & \vdots \\ \mathbf{e}_n^1 & \cdots & \mathbf{e}_n^{2t+1} \end{pmatrix},$$

which is the $t \times (2t+1)$ matrix whose rows correspond to the *honest parties' shares* of the errors, the equation above can be rewritten as

$$0 = \mathbf{G} \cdot \left(\frac{0}{\mathbf{E} \cdot \mathbf{G}^\intercal}\right),$$

but this is not possible unless $\mathbf{E} \cdot \mathbf{G}^\intercal = 0$, otherwise, $\left(\frac{0}{\mathbf{E} \cdot \mathbf{G}^\intercal}\right)$ would contain one non-zero column of weight at most t that maps to zero under $\mathbf{H}$, or in other words, this column is consistent with a polynomial of degree at most $2t$. This is not possible since this vector has at least $2t + 1$ zero entries. As a result, we see that $\mathbf{E} \cdot \mathbf{G}^\top = 0$, so the shares $\left[\mathbf{e}_j\right]_{2t}^{C_{i+1}}$ (of the honest parties) indeed are shares $[\mathbf{0}]_{2t}^{C_{i+1}}$, as desired. □

Robust Resharing from Packed to Standard Now we turn our attention to the second part of our 2-hop resharing protocol in which the parties in C_{i+1} can "unpack" the sharings they have received from C_i. In order to make this section independent of the previous one, however, we relabel the committees and assume that the committee that starts with the packed sharings is C_i, instead of C_{i+1}. In this case, the context is the following: C_i holds packed sharings $[\mathbf{x}]_{2t}^{C_i}$, and the goal is for committee C_{i+1} to obtain *unpacked* Shamir sharings $([x_1]_{2t}^{C_{i+1}}, \dots, [x_{t+1}]_{2t}^{C_{i+1}})$. We model this with Functionality $\mathcal{F}_{\text{robust-standard-reshare}}$ below. As before, the main complication in modeling this approach is that the adversary can introduce errors in the sharings distributed, which will be caught by a future committee C_{i+3}.

Figure 3.32: Functionality $\mathcal{F}_{\text{robust-standard-reshare}}$

1. Let $[\mathbf{x}]_{2t}^{C_i}$ be the packing held by the parties of C_i, corresponding to the vector of values $\mathbf{x} = (x_1, \dots, x_{t+1})$.

2. $\mathcal{F}_{\text{robust-standard-reshare}}$ first receives from the honest parties of C_i their shares of $[\mathbf{x}]_{2t}^{C_i}$ (at least $2t + 1$ of them).

3. $\mathcal{F}_{\text{robust-standard-reshare}}$ then reconstructs all of $[\mathbf{x}]_{2t}^{C_i}$ and sends the shares of corrupted parties to the adversary.

4. $\mathcal{F}_{\text{robust-standard-reshare}}$ then receives from the adversary a set of shares $\{(x_1^k, \dots, x_{t+1}^k)\}_{k \in \mathcal{T}_{C_{i+1}}}$, and errors $(e_1, \dots, e_{t+1})$.

5. Next, $\mathcal{F}_{\text{robust-standard-reshare}}$ computes the sharings $([x_1 + e_1]_{2t}^{C_{i+1}}, \dots, [x_{t+1} + e_{t+1}]_{2t}^{C_{i+1}})$ based on the t shares x_α^k from the adversary, the value $x_\alpha + e_\alpha$, and t other randomly sampled shares, for each $\alpha \in [t + 1]$, then sends to the honest parties their shares.

6. Finally, if any $e_\alpha \neq 0$, $\mathcal{F}_{\text{robust-standard-reshare}}$ sends abort to the honest parties of C_{i+3}. Otherwise, $\mathcal{F}_{\text{robust-standard-reshare}}$ asks the adversary whether to continue, and if the adversary replies (abort, A) for $A \subseteq \mathcal{H}_{C_{i+3}}$, then $\mathcal{F}_{\text{robust-standard-reshare}}$ sends abort to the honest parties A of C_{i+3}.

Our protocol to instantiate $\mathcal{F}_{\text{robust-standard-reshare}}$, Protocol $\Pi_{\text{robust-standard-reshare}}$, proceeds as follows. First, each party in C_i, the committee holding the packed sharings, secret-shares using standard Shamir secret-sharing their own share. At this point, the parties in C_{i+1} can take multiple linear combinations using Lagrange coefficients on these Shamir shares to compute shares of each one of the secrets in the original vector. As with $\Pi_{\text{robust-packed-reshare}}$, corrupt parties may attempt to change their shares when resharing. This once again is catched by employing the matrix H from Section 3.5.2.2, and the committee C_{i+3} learns whether the distributed values are consistent or not. The protocol is described in detail below.

Figure 3.33: Protocol $\Pi_{\text{robust-standard-reshare}}$

Usage: Committee C_i holds a packed sharing $[\mathbf{x}]_{2t}^{C_i}$ and committee C_{i+1} outputs standard sharings $([x_1]_{2t}^{C_{i+1}}, \dots, [x_{t+1}]_{2t}^{C_{i+1}})$ corresponding to the vector of values $\mathbf{x} = (x_1, \dots, x_{t+1})$, using committees C_{i+2} and C_{i+3} to ensure its correctness.

1. Every party P_j in committee C_i distributes degree-$2t$ standard sharings $\left[\mathbf{x}^j\right]_{2t}^{C_{i+1}}$ to committee C_{i+1} of its share $\mathbf{x}^j$ of the packed sharing $[\mathbf{x}]_{2t}^{C_i}$.

2. While the Verification phase (below) executes, the parties of committee C_{i+1} compute and output fresh shares for every $\alpha \in [t+1]$: $[x_\alpha]_{2t}^{C_{i+1}} = \sum_{j=1}^{2t+1} L_j(-\alpha) \cdot \left[\mathbf{x}^j\right]_{2t}^{C_{i+1}}$, where $L_j(-\alpha)$ are Lagrange interpolation coefficients.

Verification Phase:

1. The parties P_k of committee C_{i+1} apply the parity check matrix $\mathbf{H}$ to get $([y_1]_{2t}^{C_{i+1}}, \dots, [y_{n-2t-1}]_{2t}^{C_{i+1}})^\intercal \leftarrow \mathbf{H} \cdot (\left[\mathbf{x}^1\right]_{2t}^{C_{i+1}}, \dots, [\mathbf{x}^n]_{2t}^{C_{i+1}})^\intercal$.

2. Next, they apply super-invertible matrix $\mathbf{M}$ to get $([z_1]_{2t}^{C_{i+1}}, \ldots, [z_n]_{2t}^{C_{i+1}})^\intercal \leftarrow \mathbf{M} \cdot ([y_1]_{2t}^{C_{i+1}}, \ldots, [y_{n-2t-1}]_{2t}^{C_{i+1}})^\intercal$, and open $[z_l]_{2t}^{C_{i+1}}$ to party P_l of committee C_{i+2}.

3. Finally, each party P_l of committee C_{i+2} checks that the shares of $[z_l]_{2t}^{C_{i+1}}$ are $2t$-consistent and that they correspond to $z_l = 0$.

4. If either of the checks fail, they send abort to the parties of committee C_{i+3}.

As with $\Pi_{\text{robust-packed-reshare}}$, the communication complexity of $\Pi_{\text{robust-standard-reshare}}$ is $n \cdot n = O(n^2)$ sharings in step 1, plus $n \cdot n = O(n^2)$ sharings from second step in the verification, for a total of $O(n^2/(t+1)) = O(n)$ per value being reshared. Security is given in the following lemma.

Lemma 3.23. $\Pi_{\text{robust-standard-reshare}}$ *UC-realizes* $\mathcal{F}_{\text{robust-standard-reshare}}$.

Proof. First, we define the simulator $\mathcal{S}$:

1. $\mathcal{S}$ first receives from $\mathcal{F}_{\text{robust-standard-reshare}}$ the shares of the corrupted parties of $[\mathbf{x}]_{2t}^{C_i}$: $(\mathbf{x}^j)_{j \in \mathcal{T}_{C_i}}$

2. For each $j' \in \mathcal{H}_{C_i}$ and $k' \in \mathcal{T}_{C_{i+1}}$, $\mathcal{S}$ samples random $x_{j'k'}$ and sends it to the adversary.

3. $\mathcal{S}$ then receives from the adversary for each $j \in \mathcal{T}_{C_i}$ and $k \in \mathcal{H}_{C_{i+1}}$, value x_{jk}. For each j, $\mathcal{S}$ uses the $2t+1$ values x_{jk} to compute the sharing $\left[\widehat{\mathbf{x}^j}\right]_{2t}^{C_{i+1}} = (x_{j1}, \ldots, x_{jn})$, and the underlying secret $\widehat{\mathbf{x}^j}$ (which is a share of a packed sharing).

4. Next, $\mathcal{S}$ computes for each $j \in \mathcal{T}_{C_i}$, their error

$$e_j \leftarrow \widehat{\mathbf{x}^j} - \mathbf{x}^j,$$

and then for $\alpha \in [t+1]$, the overall errors

$$e_\alpha \leftarrow \sum_{j \in \mathcal{T}_{C_i} \cap [2t+1]} L_j(-\alpha) \cdot e_j.$$

5. $\mathcal{S}$ then computes for $\alpha \in [2t+1], k' \in \mathcal{T}_{C_{i+1}}, x_\alpha^{k'} = \sum_{j=1}^{2t+1} L_j(-\alpha) \cdot x_{jk'}$ and sends these values along with $e_1, \ldots, e_{t+1}$ to $\mathcal{F}_{\text{robust-standard-reshare}}$.

6. Next, $\mathcal{S}$ computes the matrix-vector product

$$(y_1, \ldots, y_{n-2t-1})^\intercal \leftarrow \mathbf{H} \cdot (e_1, \ldots, e_n)^\intercal,$$

   where $e_j = 0$ for $j \in \mathcal{H}_{C_i}$

7. $\mathcal{S}$ also computes for each $k' \in \mathcal{T}_{C_{i+1}}$:

$$(y_1^{k'}, \ldots, y_{n-2t-1}^{k'})^\intercal \leftarrow \mathbf{H} \cdot (x_{1k'}, \ldots, x_{nk'})^\intercal.$$

8. Additionally, $\mathcal{S}$ for the first t parties $k \in \mathcal{H}_{C_{i+1}}$ samples random $(y_1^k, \ldots, y_{n-2t-1}^k)$.

9. For each $\mu \in [n-2t-1]$, using y_μ and the shares $y_\mu^{k'}$ of $k' \in \mathcal{T}_{C_{i+1}}$ and y_μ^k of the first t parties $k \in \mathcal{H}_{C_{i+1}}$ (i.e., $2t+1$ total degrees of freedom), $\mathcal{S}$ computes the sharing $\left[y_\mu\right]_{2t}^{C_{i+1}}$.

10. Next, $\mathcal{S}$ computes $([z_1]_{2t}^{C_{i+1}}, \ldots, [z_n]_{2t}^{C_{i+1}}) \leftarrow \mathbf{M} \cdot ([y_1]_{2t}^{C_{i+1}}, \ldots, [y_{n-2t-1}]_{2t}^{C_{i+1}})$, and sends the honest parties' shares of $[z_l]_{2t}^{C_{i+1}}$ to each corrupted party P_l of committee C_{i+2}.

11. For each honest party P_l of committee C_{i+2}, $\mathcal{S}$ receives from the adversary: committee C_{i+1} corrupted parties' shares of $[z_l]_{2t}^{C_{i+1}}$. If these shares do not match what $\mathcal{S}$ had already computed, or the underlying computed value $z_l \neq 0$, $\mathcal{S}$ sends abort on behalf of P_l to all corrupted parties of committee C_{i+3}.

12. Finally, if $\mathcal{S}$ sent any abort in the above step, it sets $A \leftarrow \mathcal{H}_{C_{i+3}}$; otherwise, it sets A to be those honest parties that receive from any corrupt party abort. Then, when $\mathcal{S}$ is asked by $\mathcal{F}_{\text{robust-standard-reshare}}$ whether to continue, it replies with (abort, A).

Now we show that the real world and ideal world are identically distributed to the adversary. In Step 1 of $\Pi_{\text{robust-standard-reshare}}$, the adversary receives the t committee C_{i+1} corrupted parties' shares of the committee C_i honest parties' sharings $\left[\mathbf{x}^j\right]_{2t}^{C_{i+1}}$. Since these are degree-$2t$ standard Shamir sharings, there are $2t$ remaining (random) degrees of freedom in defining the underlying polynomial, and thus the t shares that the adversary sees are uniformly random, while leaving t remaining (random) degrees of freedom for each sharing, or $t \cdot (2t+1)$ in total. Indeed, Step 2 of $\mathcal{S}$ is to send uniformly random shares to the adversary, corresponding to the sharings, and thus the view of the adversary is identical in the real and ideal worlds for this step.

In Step 2 of $\Pi_{\text{robust-standard-reshare}}$, the honest parties output for $\alpha \in [t+1]$ their shares $[x_\alpha]_{2t}^{C_{i+1}} = \sum_{j=1}^{2t+1} L_j(-\alpha) \cdot \left[\mathbf{x}^j\right]_{2t}^{C_{i+1}}$ based on their own as well as possibly some corrupted sharings $\left[\mathbf{x}^j\right]_{2t}^{C_{i+1}}$. The honest parties' shares uniquely define a degree-$2t$ polynomial for $\alpha \in [t+1]$, which evaluated on 0 gives

$$\sum_{k \in \mathcal{H}_{C_{i+1}}} L_k(0) \sum_{j=1}^{2t+1} L_j(-\alpha) \cdot x_{jk} = \sum_{j=1}^{2t+1} L_j(-\alpha) \sum_{k \in \mathcal{H}_{C_{i+1}}} L_k(0) x_{jk} = \sum_{j=1}^{2t+1} L_j(-\alpha) \widehat{\mathbf{x}^j}$$

$$= \sum_{j \in \mathcal{H}_{C_{i+1}} \cap [2t+1]} L_j(-\alpha) \cdot \mathbf{x}^j + \sum_{j \in \mathcal{T}_{C_{i+1}} \cap [2t+1]} L_j(-\alpha) \cdot (\mathbf{x}^j + e_j) = x_\alpha + \sum_{j \in \mathcal{T}_{C_{i+1}} \cap [2t+1]} L_j(-\alpha) \cdot e_j,$$

where e_j is some error injected by the adversary for corrupt party P_j. Similarly, for $k' \in \mathcal{T}_{C_{i+1}}$, the polynomial evaluated on k', i.e., $P_{k'}$'s share, gives

$$\sum_{j=1}^{2t+1} L_j(-\alpha) \sum_{k \in \mathcal{H}_{C_{i+1}}} L_k(k') x_{jk} = \sum_{j=1}^{2t+1} L_j(-\alpha) x_{jk'},$$

where $x_{jk'}$ is the corrupted party $P_{k'}$'s share of $\left[\mathbf{x}^j\right]_{2t}^{C_{i+1}}$. In the ideal world, $\mathcal{S}$ in Step 3 for $j \in \mathcal{T}_{C_i}$ reconstructs the entire sharings $\left[\mathbf{x}^j\right]_{2t}^{C_{i+1}}$. This allows $\mathcal{S}$ to reconstruct the corresponding corrupted party $P_{k'}$'s shares of each of these sharings, $x_{jk'}$, as well as the underlying $\widehat{\mathbf{x}^j}$, which is supposed to be P_j's share of the original packed sharing. The former allows it to, along with the corrupted party's shares of honest party's sharings $\left[\mathbf{x}^j\right]_{2t}^{C_{i+1}}$, compute the corrupted parties' shares of $[x_\alpha]_{2t}^{C_{i+1}}$, for $\alpha \in [t+1]$ which correspond to exactly those in the real world, as computed above. The latter allows it to compute the error e_j of these shares, as compared to those received from $\mathcal{F}_{\text{robust-standard-reshare}}$ in Step 1, and from this, the overall errors $e_1, \ldots, e_{t+1}$, which is exactly the error in the real world, as computed above. Finally, $\mathcal{S}$, gives to $\mathcal{F}_{\text{robust-standard-reshare}}$, for $\alpha \in [t+1]$, the corrupted party's shares of $[x_\alpha]_{2t}^{C_{i+1}}$, and the error e_α, from which $\mathcal{F}_{\text{robust-standard-reshare}}$ further computes the perturbed underlying values $x_\alpha + e_\alpha$. This gives $\mathcal{F}_{\text{robust-standard-reshare}}$ $t+1$ points on degree-$2t$ polynomials, which are distributed the same as in the real world, and based on this and t other randomly sampled points on each polynomial, $\mathcal{F}_{\text{robust-standard-reshare}}$ reconstructs $[x_\alpha]_{2t}^{C_{i+1}}$ and gives honest parties their shares. Note that we had $2t^2 + t$ total degrees of freedom above and this uses up only $t \cdot (t+1)$ of them. Thus, the shares output by the honest parties in the real and ideal world are distributed identically. Indeed, for the first $t+1$ honest parties $\alpha \in \mathcal{H}_{C_i}$, we can use the t remaining random shares of $[\mathbf{x}^\alpha]_{2t}^{C_{i+1}}$, $x_{\alpha 1}, \ldots, x_{\alpha t}$ to randomly perturb the shares of $[x_\alpha]_{2t}^{C_{i+1}}$ of the first $t+1$ honest parties of $\mathcal{H}_{C_{i+1}}$, as desired. Moreover, we have t^2 remaining (random) degrees of freedom, corresponding to the t remaining random shares of $\left[\mathbf{x}^j\right]_{2t}^{C_{i+1}}$, for the last $t+1$ honest parties $j \in \mathcal{H}_{C_i}$.

In Step 2 of the **Verification Phase** of $\Pi_{\text{robust-standard-reshare}}$, the adversary receives on behalf of corrupt party P_l of committee C_{i+2}, the honest parties' shares of $[z_l]_{2t}^{C_{i+1}}$. These shares come from them computing

$$\mathbf{M} \cdot \mathbf{H} \cdot \left(\left[\widehat{\mathbf{x}^1}\right]_{2t}^{C_{i+1}}, \ldots, \left[\widehat{\mathbf{x}^n}\right]_{2t}^{C_{i+1}}\right)^\intercal =$$

$$\mathbf{M} \cdot \mathbf{H} \cdot \left(\left(\left[\mathbf{x}^1\right]_{2t}^{C_{i+1}}, \ldots, [\mathbf{x}^n]_{2t}^{C_{i+1}}\right)^\intercal + \left([e_1]_{2t}^{C_{i+1}}, \ldots, [e_n]_{2t}^{C_{i+1}}\right)^\intercal\right),[12]$$

[12]Such decomposition of the maliciously shared $\left[\widehat{\mathbf{x}^j}\right]_{2t}^{C_{i+1}}$ is always possible since we can define some canonical packed sharing $\left[\mathbf{x}^j\right]_{2t}^{C_{i+1}} := (x_{j1}, \ldots, x_{jn})$, and $\left[e_j\right]_{2t}^{C_{i+1}}$ as the rest.

where $\left[e_j\right]_{2t}^{C_{i+1}} = 0$ for $j \in \mathcal{H}_{C_i}$

$$= ([0]_{2t}^{C_{i+1}}, \ldots, [0]_{2t}^{C_{i+1}})^\top + \mathbf{M} \cdot \mathbf{H} \cdot \left([e_1]_{2t}^{C_{i+1}}, \ldots, [e_n]_{2t}^{C_{i+1}}\right)^\top,$$

where the identity of the first term is due to the fact that the underlying shares $\mathbf{x}^1, \ldots, \mathbf{x}^n$ of the respective sharings $\left[\mathbf{x}^j\right]_{2t}^{C_{i+1}}$ indeed correspond to valid points of a polynomial of degree $\leq 2t$, and therefore applying H to them results in zeroes in each element of the output vector.

In the ideal world, $\mathcal{S}$ computes $\mathbf{H} \cdot (e_1, \ldots, e_n)^\top$ and the corrupted parties' shares of $([y_1]_{2t}^{C_{i+1}}, \ldots, [y_{n-2t-1}]_{2t}^{C_{i+1}})^\top \leftarrow \mathbf{H} \cdot \left(\left[\widehat{\mathbf{x}^1}\right]_{2t}^{C_{i+1}}, \ldots, \left[\widehat{\mathbf{x}^n}\right]_{2t}^{C_{i+1}}\right)^\top$. This gives $\mathcal{S}$: $t+1$ evaluations of the polynomials underlying $([y_1]_{2t}^{C_{i+1}}, \ldots, [y_{n-2t-1}]_{2t}^{C_{i+1}})$; from which, with t more random evaluations, it can reconstruct the whole sharings. Recall that we had t^2 remaining degrees of freedom, and this is exactly t^2 random values, since $n - 2t - 1 = t$, so this results in an identical distribution as the real world. Indeed, for the last $t+1$ honest parties $j \in \mathcal{H}_{C_i}$, we can use the t remaining random shares of $\left[\mathbf{x}^j\right]_{2t}^{C_{i+1}}$, $x_{j1}, \ldots, x_{jt}$ to randomly perturb the first t shares of $\left[y_j\right]_{2t}^{C_{i+1}}$, respectively, as desired. It can then compute

$$([z_1]_{2t}^{C_{i+1}}, \ldots, [z_n]_{2t}^{C_{i+1}})^\top \leftarrow \mathbf{M} \cdot ([y_1]_{2t}^{C_{i+1}}, \ldots, [y_{n-2t-1}]_{2t}^{C_{i+1}})^\top,$$

from which it can send the corrupted parties P_l all of the honest parties' shares of $[z_l]_{2t}^{C_{i+1}}$. Thus, the real world and ideal world are identically distributed at this step.

In step 2 of the **Verification Phase** of $\Pi_{\text{robust-standard-reshare}}$, if the corrupted parties' shares of $[z_l]_{2t}^{C_{i+1}}$ are not consistent with those of the honest parties, then we know from the error-detection properties of packed sharings that honest party P_l will send abort to the parties of committee C_{i+3}, and similarly if $z_l \neq 0$. In the ideal world, $\mathcal{S}$ in Step 11, first checks if the corrupted parties' shares of $[z_l]_{2t}^{C_{i+1}}$, for honest party P_l of C_{i+2}, match that which it had already computed (i.e., are consistent with those of honest parties), and then also if $z_l = 0$. If either of these checks fail, $\mathcal{S}$ sends abort on behalf of P_l to all corrupted parties of committee C_{i+3}. Thus, the real world and ideal world are identically distributed at this step.

Finally, we need to show that if any $e_\alpha \neq 0$, then the honest parties abort in the real world (as they are forced to in the ideal world). We do so by showing the contrapositive: i.e., if the honest parties in the real world do not abort, then each $e_\alpha = 0$. If the honest parties do not abort, then it must be that for each of the $2t+1$ honest parties P_l in committee C_{i+2}, $[z_l]_{2t}^{C_{i+1}}$ are $2t$-consistent and correspond to $z_l = 0$. By the error-correction properties of super-invertible matrix M, this must also mean that $y_1 = \cdots = y_{n-2t-1} = 0$ (because any $n - 2t - 1 \leq 2t + 1$ of the honest codeword symbols can be multiplied by the inverse of the corresponding submatrix of M to get back the original message, which therefore must be all zeroes). As written above, we have that $([y_1]_{2t}^{C_{i+1}}, \ldots, [y_{n-2t-1}]_{2t}^{C_{i+1}})^\intercal =$

$$\mathbf{H} \cdot \left(\left(\left[\mathbf{x}^1\right]_{2t}^{C_{i+1}}, \ldots, [\mathbf{x}^n]_{2t}^{C_{i+1}} \right)^\intercal + \left([e_1]_{2t}^{C_{i+1}}, \ldots, [e_n]_{2t}^{C_{i+1}} \right)^\intercal \right),$$

where $\left[e_j\right]_{2t}^{C_{i+1}} = 0$ for $j \in \mathcal{H}_{C_i}$

$$= ([0]_{2t}^{C_{i+1}}, \ldots, [0]_{2t}^{C_{i+1}})^\intercal + \mathbf{H} \cdot \left([e_1]_{2t}^{C_{i+1}}, \ldots, [e_n]_{2t}^{C_{i+1}} \right)^\intercal .$$

Assume w.l.o.g., now that the first $2t+1$ parties are honest, so that the above is equal to:

$$= ([0]_{2t}^{C_{i+1}}, \ldots, [0]_{2t}^{C_{i+1}})^\intercal + \mathbf{H} \cdot \left(0, \ldots, 0, [e_{2t+2}]_{2t}^{C_{i+1}}, \ldots, [e_n]_{2t}^{C_{i+1}} \right)^\intercal .$$

Let us denote the matrix of shares of the sharings $([y_1]_{2t}^{C_{i+1}}, \ldots, [y_{n-2t-1}]_{2t}^{C_{i+1}})$ as $\mathbf{Y}$ (each column is the set of shares held by each party). Then,

$$\mathbf{Y} = \begin{pmatrix} y_1^1 & \cdots & y_1^n \\ \vdots & \ddots & \vdots \\ y_{n-2t-1}^1 & \cdots & y_{n-2t-1}^n \end{pmatrix} = \begin{pmatrix} 0_1^1 & \cdots & 0_1^n \\ \vdots & \ddots & \vdots \\ 0_{n-2t-1}^1 & \cdots & 0_{n-2t-1}^n \end{pmatrix} + \mathbf{H} \cdot \begin{pmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \\ e_{2t+2}^1 & \cdots & e_{2t+2}^n \\ \vdots & \ddots & \vdots \\ e_n^1 & \cdots & e_n^n \end{pmatrix}.$$

Testing if a vector $\mathbf{x} \in \mathbb{F}^{2t+1}$ corresponds to a degree-$2t$ sharing of 0 can be done by interpolating a polynomial f of degree $\leq 2t$ from the $2t+1$ entries, and checking that $f(0) = 0$. This can be directly expressed as an inner product $0 = \mathbf{g}^\top \cdot \mathbf{x}$, where $\mathbf{g}$ is a $(2t+1)$-dimensional vector. Since each y_j is equal to 0, the honest entries of each row of Y, multiplied by $\mathbf{g}$, gives zero, or in other words:

$$0 = \begin{pmatrix} y_1^1 & \cdots & y_1^n \\ \vdots & \ddots & \vdots \\ y_{n-2t-1}^1 & \cdots & y_{n-2t-1}^n \end{pmatrix} \cdot \begin{pmatrix} \mathbf{g} \\ \hline 0 \end{pmatrix} = \overbrace{\begin{pmatrix} 0_1^1 & \cdots & 0_1^n \\ \vdots & \ddots & \vdots \\ 0_{n-2t-1}^1 & \cdots & 0_{n-2t-1}^n \end{pmatrix} \cdot \begin{pmatrix} \mathbf{g} \\ \hline 0 \end{pmatrix}}^{0}$$
$$+ \mathbf{H} \cdot \begin{pmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \\ e_{2t+2}^1 & \cdots & e_{2t+2}^n \\ \vdots & \ddots & \vdots \\ e_n^1 & \cdots & e_n^n \end{pmatrix} \cdot \begin{pmatrix} \mathbf{g} \\ \hline 0 \end{pmatrix}$$

Denoting

$$\mathbf{E} = \begin{pmatrix} e^1_{2t+2} & \cdots & e^{2t+1}_{2t+2} \\ \vdots & \ddots & \vdots \\ e^1_n & \cdots & e^{2t+1}_n \end{pmatrix},$$

which is the $t \times (2t+1)$ matrix whose rows correspond to the *honest parties' shares* of the errors, the equation above can be rewritten as

$$0 = \mathbf{H} \cdot \left(\frac{0}{\mathbf{E} \cdot \mathbf{g}}\right),$$

but this is not possible unless $\mathbf{E} \cdot \mathbf{g} = 0$, otherwise, $\left(\frac{0}{\mathbf{E} \cdot \mathbf{g}}\right)$ would contain one non-zero column of weight at most t that maps to zero under $\mathbf{H}$, or in other words, this column is consistent with a polynomial of degree at most $2t$. This is not possible since this vector has at least $2t+1$ zero entries. As a result, we see that $\mathbf{E} \cdot \mathbf{g} = 0$, so the shares $\left[e_j\right]^{C_{i+1}}_{2t}$ (of the honest parties) indeed are shares $[0]^{C_{i+1}}_{2t}$, as desired. □

3.5.2.3 Two-Thirds Honest Majority Main Protocol

With our core construction for linear communication resharing at hand, given by protocols $\Pi_{\text{robust-packed-reshare}}$ and $\Pi_{\text{robust-standard-reshare}}$ from Section 3.5.2.2, we are ready to present in detail our end-to-end Fluid MPC protocol with linear communication, for perfect security with abort. First, in Section 3.5.2.3 we describe how multiplications are handled. Then, in Section 3.5.2.3 we present our actual Fluid MPC protocol.

Linear-Overhead Multiplication Procedure In our MPC protocol $\Pi_{\text{main-2/3-hm}}$ from Section 3.5.2.3, the parties in a committee C_{i+1} will hold two groups of sharings $([x_1]^{C_{i+1}}_{2t}, , \ldots, [x_{t+1}]^{C_{i+1}}_{2t})$ and $([y_1]^{C_{i+1}}_{2t}, \ldots, [y_{t+1}]^{C_{i+1}}_{2t})$, and the goal will be for the parties in C_{i+3} to obtain sharings $([x_1 y_1]^{C_{i+3}}_{2t},$

$\dots, [x_{t+1}y_{t+1}]_{2t}^{C_{i+3}}$). This specific step in the protocol is addressed by Procedure π_{mult}, which we describe below, and is caled from within Protocol $\Pi_{\mathsf{main\text{-}2/3\text{-}hm}}$.

At a high-level, committees C_{i+1} through committees C_{i+3} will execute beaver multiplication [Beaver 1992], using multiple kings in C_{i+2}, as specified by the standard technique of [Damgård and Nielsen 2007]. To do this, committee C_i first invokes $\mathcal{F}_{\mathsf{rand\text{-}2/3\text{-}hm}}$ to output random sharings $[a_\alpha]_t^{C_{i+1}}, [b_\alpha]_t^{C_{i+1}}$, for $\alpha \in [t+1]$ to committee C_{i+1}. Committee C_{i+1} then locally computes for each $\alpha \in [t+1]$, $[c_\alpha]_{2t}^{C_{i+1}} \leftarrow [a_\alpha]_t^{C_{i+1}} \cdot [b_\alpha]_t^{C_{i+1}}$. Then, committee C_{i+1} invokes $\mathcal{F}_{\mathsf{robust\text{-}packed\text{-}reshare}}$ on these sharings so that committee C_{i+2} receives packing $[\mathbf{a}]_{2t}^{C_{i+2}}$ for vector $\mathbf{a} = (a_1, \dots, a_{t+1})$, and the same for $[\mathbf{b}]_{2t}^{C_{i+2}}, [\mathbf{c}]_{2t}^{C_{i+2}}$. Finally, committee C_{i+3} invokes $\mathcal{F}_{\mathsf{robust\text{-}standard\text{-}reshare}}$ on these packings, so that committee C_{i+3} receives $[a_\alpha]_{2t}^{C_{i+3}}, [b_\alpha]_{2t}^{C_{i+3}}, [c_\alpha]_{2t}^{C_{i+3}}$ for $\alpha \in [t+1]$, where $(a_\alpha, b_\alpha, c_\alpha)$ are the same triples that committee C_{i+1} had.

Note that the triple used by the parties in C_{i+1} is $([a_\alpha]_t^{C_{i+1}}, [b_\alpha]_t^{C_{i+1}}, [a_\alpha]_t^{C_{i+1}} \cdot [b_\alpha]_t^{C_{i+1}})$, which is *not* a truly random triple of degree-$2t$ since (1) the first two entries are random of degree-t, not $2t$, and (2) the the underlying polynomial in the last entry is not random as it is the product of two degree-t polynomials. However, this is acceptable in our context since these sharings will be *reshared* with $\mathcal{F}_{\mathsf{robust\text{-}packed\text{-}reshare}}$ and $\mathcal{F}_{\mathsf{robust\text{-}standard\text{-}reshare}}$, which are agnostic to the distribution of the input sharings, and guarantee the output sharings are freshly random, as required.

Figure 3.34: Procedure π_{mult}

Usage: Multiply $[x_\alpha]_{2t}^{C_{i+1}}$ and $[y_\alpha]_{2t}^{C_{i+1}}$ held by committee C_{i+1} so that committee C_{i+3} outputs $[x_\alpha y_\alpha]_{2t}^{C_{i+1}}$, for $\alpha \in [t+1]$.

1. Committee C_i first invokes $\mathcal{F}_{\mathsf{rand\text{-}2/3\text{-}hm}}$ to output random sharings $[a_\alpha]_t^{C_{i+1}}, [b_\alpha]_t^{C_{i+1}}$, for $\alpha \in [t+1]$ to committee C_{i+1} (using committees C_{i+2} and C_{i+3} for verification, which aborts if needed).

2. The parties P_j in C_{i+1} next multiply for each $\alpha \in [t+1]$, $[c_\alpha]_{2t}^{C_{i+1}} \leftarrow [a_\alpha]_t^{C_{i+1}} \cdot [b_\alpha]_t^{C_{i+1}}$.

3. Committee C_{i+1} then computes $[d_\alpha]_{2t}^{C_{i+1}} \leftarrow [x_\alpha]_{2t}^{C_{i+1}} + [a_\alpha]_t^{C_{i+1}}$ and $[e_\alpha]_{2t}^{C_{i+1}} \leftarrow [y_\alpha]_{2t}^{C_{i+1}} +$

$[b_\alpha]_t^{C_{i+1}}$ for every $\alpha \in [t+1]$.

4. Committee C_{i+1} next applies super-invertible matrix $\mathbf{M}$ to $([d_1]_{2t}^{C_{i+1}}, \ldots, [d_{t+1}]_{2t}^{C_{i+1}})$ to get $\left([d'_1]_{2t}^{C_{i+1}}, \ldots, [d'_n]_{2t}^{C_{i+1}}\right)$, and $([e_1]_{2t}^{C_{i+1}}, \ldots, [e_{t+1}]_{2t}^{C_{i+1}}$ to get $\left([e'_1]_{2t}^{C_{i+1}}, \ldots, [e'_n]_{2t}^{C_{i+1}}\right)$ and opens $[d'_k]_{2t}^{C_{i+1}}, [e'_k]_{2t}^{C_{i+1}}$ to party P_k of committee C_{i+2}.

5. Simultaneously, the parties P_j in C_{i+1} then invoke $\mathcal{F}_{\text{robust-packed-reshare}}$ towards committee C_{i+2} on input standard sharings $([a_1]_t^{C_{i+2}}, \ldots, [a_{t+1}]_t^{C_{i+2}})$, so that C_{i+2} gets the packing $[\mathbf{a}]_{2t}^{C_{i+2}}$ for vector $\mathbf{a} = (a_1, \ldots, a_{t+1})$ (using committees C_{i+3} and C_{i+4} for verification, which aborts if needed).

6. They do the same for $([b_1]_t^{C_{i+2}}, \ldots, [b_{t+1}]_t^{C_{i+2}})$ and $([c_1]_{2t}^{C_{i+2}}, \ldots, [c_{t+1}]_{2t}^{C_{i+2}})$ to get packings $[\mathbf{b}]_{2t}^{C_{i+2}}$ and $[\mathbf{c}]_{2t}^{C_{i+2}}$ for vectors $\mathbf{b} = (b_1, \ldots, b_{t+1})$ and $\mathbf{c} = (c_1, \ldots, c_{t+1})$, respectively.

7. The parties P_k of C_{i+2} then reconstruct d'_k, e'_k (or abort if unsuccessful) and send them to all parties of committee C_{i+3}.

8. Simultaneously, they run $\mathcal{F}_{\text{robust-standard-reshare}}$ towards committee C_{i+3} on input $[\mathbf{a}]_{2t}^{C_{i+2}}$ so that C_{i+3} gets the standard sharings $([a_1]_{2t}^{C_{i+3}}, \ldots, [a_{t+1}]_{2t}^{C_{i+3}})$ (using committees C_{i+4} and C_{i+5} for verification, which aborts if needed). They do the same with $[\mathbf{b}]_{2t}^{C_{i+2}}$ and $[\mathbf{c}]_{2t}^{C_{i+2}}$.

9. The parties of C_{i+3} run Berlekamp-Welch on $(d'_1, \ldots, d'_n)$ and $(e'_1, \ldots, e'_n)$ to get $\{d_\alpha\}_{\alpha\in[t+1]}$ and $\{e_\alpha\}_{\alpha\in[t+1]}$, respectively.

10. Finally, committee C_{i+3} outputs $[x_\alpha y_\alpha]_{2t}^{C_{i+3}} = d_\alpha \cdot e_\alpha - d_\alpha \cdot [b_\alpha]_{2t}^{C_{i+3}} - e_\alpha \cdot [a_\alpha]_{2t}^{C_{i+3}} + [c_\alpha]_{2t}^{C_{i+3}}$, for each $\alpha \in [t+1]$.

The main properties of Procedure π_{mult} that we will use are summarized in Lemmas 3.24 and 3.25 below, whose proof is given below. We will use this Lemma in the proof of Theorem 3.27, which proves the security of our MPC protocol $\Pi_{\text{main-2/3-hm}}$.

Lemma 3.24. *Let it be the case that for every* $\alpha \in [t+1]$, *either the adversary completely knows* *the sharing $[x_\alpha]_{2t}^{C_{i+1}}$, or the first t honest parties' shares are distributed randomly to the adversary. If given in addition to the adversary's known shares: random a_α^* for $\alpha \in [t+1]$, and also for those $\alpha \in [t+1]$ satisfying the latter case, random $x_\alpha^1, \ldots, x_\alpha^t$, then values distributed identically to the adversary to honest parties' shares of $\left[d_k'\right]_{2t}^{C_{i+1}}$ received by P_k of C_{i+2}, for $k \in \mathcal{T}_{C_{i+2}}$, in π_{mult} can be determined. Moreover, values distributed identically to the adversary to d_k' sent from honest party P_k of C_{i+2}, for $k \in \mathcal{H}_{C_{i+2}}$, in π_{mult} can be determined. The same holds for the corresponding values and sharings of e_k' for $k \in [n]$.*

Proof. Assume w.l.o.g., that the first $2t+1$ parties are honest. From the definition of $\mathcal{F}_{\mathsf{rand\text{-}hm}}$, we can conclude that the shares $(a_1^{2t+1}, \ldots, a_{t+1}^{2t+1}) = (a_1^*, \ldots, a_{t+1}^*)$ of the last honest party P_{2t+1} of sharings $([a_1]_t^{C_{i+1}}, \ldots, [a_{t+1}]_t^{C_{i+1}})$, respectively, received in Step 1 of π_{mult} are distributed uniformly at random to the adversary. Now, let us analyze the first case that for some x_α, $\alpha \in [t+1]$, sharing $[x_\alpha]_{2t}^{C_{i+1}}$ is completely known to the adversary. Then, given random a_α^{2t+1}, the t corrupted parties' shares of $[a_\alpha]_t^{C_{i+1}}$, and the error sharing $[e_\alpha]_{t'}^{C_{i+1}}$; the first $2t+1$ honest parties' shares $a_\alpha^1 + e_\alpha^1, \ldots, a_\alpha^{2t+1} + e_\alpha^{2t+1}$ can be determined. Therefore, given random a_α^{2t+1}, and the shares $x_\alpha^1, \ldots, x_\alpha^{2t+1}$ that the adversary knows, $x_\alpha^1 + a_\alpha^1 + e_\alpha^1, \ldots, x_\alpha^{2t+1} + a_\alpha^{2t+1} + e_\alpha^{2t+1}$ can all be determined.

The other case is that for x_α, the adversary only knows the corrupted parties' shares of sharing $[x_\alpha]_{2t}^{C_{i+1}}$. This means that $x_\alpha^1, \ldots, x_\alpha^t$ are distributed randomly to the adversary, in addition to a_α^{2t+1}. Therefore, all of $x_\alpha^1 + a_\alpha^1, \ldots, x_\alpha^t + a_\alpha^t$ and $x_\alpha^{2t+1} + a_\alpha^{2t+1}$ are distributed randomly to the adversary. Given these $t+1$ random shares and the t corrupted parties' shares of $[x_\alpha + a_\alpha]_t^{C_{i+1}}$, the remaining shares $x_\alpha^{t+1} + a_\alpha^{t+1}, \ldots, x_\alpha^{2t} + a_\alpha^{2t}$ can be determined.

Putting the observations of the above two cases together, for all $\alpha \in [t+1]$, given some values that are uniformly random to the adversary, the shares of $[x_\alpha + a_\alpha]_{2t}^{C_{i+1}}$ of the honest parties can be determined. Therefore, so too can the honest parties' shares of

$$\left(\left[d_1'\right]_{2t}^{C_{i+1}}, \ldots, \left[d_n'\right]_{2t}^{C_{i+1}}\right)^\top \leftarrow \mathbf{M} \cdot \left([x_1 + a_1]_{2t}^{C_{i+1}}, \ldots, [x_{t+1} + a_{t+1}]_{2t}^{C_{i+1}}\right)^\top.$$

It then also trivially follows that the values $d'_1, \ldots, d'_n$ can be determined.

It is clear that the same argument can be applied for the values and sharings of $e'_1, \ldots, e'_n$. $\square$

Lemma 3.25. *If no parties of committee C_{i+3} receive* abort *in π_{mult}, then they always correctly determine the values $(x_1 + a_1, \ldots, x_{t+1} + a_{t+1})$ and $(y_1 + b_1, \ldots, y_{t+1} + b_{t+1})$.*

Proof. This follows directly from the error-detection of reconstructions and linearity of degree-$2t$ Shamir secret sharings, as well as the the error-correcting properties of super-invertible matrix $\mathbf{M}$. In particular, if none of the parties abort, then by the aforementioned properties of Shamir secret sharings, the honest parties of committee C_{i+2} reconstruct the k-th symbol d'_k of the codeword $(d'_1, \ldots, d'_n)$, for $k \in \mathcal{H}_{C_{i+2}}$, of message $(d_1, \ldots, d_{t+1}) = (x_1 + a_1, \ldots, x_{t+1} + a_{t+1})$, and forward them to the parties of committee C_{i+3}. The parties of committee C_{i+3} then receive the whole codeword, of which up to t symbols (received from the corrupted parties) are erroneous. Therefore, by the error-correcting properties of super-invertible matrix $\mathbf{M}$, the Berlekamp-Welch algorithm will recover the correct original message $(d_1, \ldots, d_{t+1})$ for the parties. The same clearly applies for $(e_1, \ldots, e_{t+1})$. $\square$

Main Protocol At this point we are finally ready to present our main Fluid MPC protocol with linear communication overhead and perfect security with abort, $\Pi_{\mathsf{main\text{-}2/3\text{-}hm}}$. For each input $x \in \mathbb{F}$ to the computation, the clients simply invoke $\mathcal{F}_{\mathsf{input}}$ on x. Then, for the execution phase, for every batch of multiplication gates at a given layer, the current committee C_i runs π_{mult}, so that committee C_{i+2} receives sharings of the products. For every batch of identity gates at the layer, committee C_i invokes $\mathcal{F}_{\mathsf{robust\text{-}packed\text{-}reshare}}$ on the input sharings, and then committee C_{i+1} invokes $\mathcal{F}_{\mathsf{robust\text{-}standard\text{-}reshare}}$ on the packed secret sharing received from $\mathcal{F}_{\mathsf{robust\text{-}packed\text{-}reshare}}$, so that C_{i+2} receives standard sharings of the inputs to the identity gates. For every batch of addition gates, committee C_i first adds the sharings for each gate together, then proceeds exactly as described for identity gates above. Finally, for each output z, the parties of the last committee C_ℓ and the clients

C_{clnt} together invoke $\mathcal{F}_{\mathsf{output}}$ on the sharing $[z]_{2t}^{C_\ell}$. The protocol is described in detail below, and its security is proven right after.

Figure 3.35: Protocol $\Pi_{\mathsf{main\text{-}2/3\text{-}hm}}$

Input Phase:

1. The clients invoke $\mathcal{F}_{\mathsf{input}}$ on each input x and receive back either abort or shares of $[x]_{2t}^{C_{\mathsf{clnt}}}$.

Execution Phase: Every other committee (with the help of the others) will compute the gates at each layer of the circuit as below:

Identity Gates: To forward $([x_1]_{2t}^{C_i}, \ldots, [x_{t+1}]_{2t}^{C_i})$:

1. The parties of committee C_i invoke $\mathcal{F}_{\mathsf{robust\text{-}packed\text{-}reshare}}$ towards committee C_{i+1} on the above sharings, so that C_{i+1} gets the packing $[\mathbf{x}]_{2t}^{C_i}$ for vector $\mathbf{x} = (x_1, \ldots, x_{t+1})$ (using committees C_{i+2} and C_{i+3} for verification, which aborts if needed).

2. Then, the parties of committee C_{i+1} invoke $\mathcal{F}_{\mathsf{robust\text{-}standard\text{-}reshare}}$ towards committee C_{i+2} on input $[\mathbf{x}]_{2t}^{C_i}$, so that C_{i+2} gets the standard sharings $([x_1]_{2t}^{C_{i+2}}, \ldots, [x_{t+1}]_{2t}^{C_{i+2}})$ (using committees C_{i+3} and C_{i+4} for verification, which aborts if needed).

Addition: To component-wise add (and forward) $([x_1]_{2t}^{C_i}, \ldots, [x_{t+1}]_{2t}^{C_i})$ and $([y_1]_{2t}^{C_i}, \ldots, [y_{t+1}]_{2t}^{C_i})$, the parties of committee C_i first directly compute

$$([x_1]_{2t}^{C_i} + [y_1]_{2t}^{C_i}, \ldots, [x_{t+1}]_{2t}^{C_i} + [y_{t+1}]_{2t}^{C_i}),$$

then run the identity gate procedure on these sharings.

Multiplication: To component-wise multiply $([x_1]_{2t}^{C_i}, \ldots, [x_{t+1}]_{2t}^{C_i})$ and $([y_1]_{2t}^{C_i}, \ldots, [y_{t+1}]_{2t}^{C_i})$, the parties of Committee C_i run π_{mult} on them so that the parties of committee C_{i+2} receive $([x_1 y_1]_{2t}^{C_{i+2}}, \ldots, [x_{t+1} y_{t+1}]_{2t}^{C_{i+2}})$.

Output Phase:

1. For each output gate belonging to a client, the parties of committee C_ℓ invoke $\mathcal{F}_{\mathsf{output}}$ on the corresponding sharing $[z]_{2t}^{C_\ell}$.

2. If the client receives abort from $\mathcal{F}_{\text{output}}$, then it sends abort to all other clients and aborts itself.

3. The clients then wait until the verification phases of the procedures of the execution phase have ended to output their values z received from $\mathcal{F}_{\text{output}}$.

Per group of $t+1$ multiplication gates, the communication complexity of $\Pi_{\text{main-2/3-hm}}$ is that of π_{mult}, which is the sum of the complexities of $\Pi_{\text{rand-2/3-hm}}$ (step 1), $\Pi_{\text{robust-packed-reshare}}$ (step 5) and $\Pi_{\text{robust-standard-reshare}}$ (step 8). This is all $O(n^2)$, which is a total of $O(n^2/(t+1)) = O(n)$ per multiplication gate.

Before we prove the security of our protocol, we consider the following simple Lemma that will prove handy.

Lemma 3.26. *If $\mathcal{F}_{\text{robust-packed-reshare}}$ sends the parties of committee C_{i+4}* abort*, then they abort. Similarly, if $\mathcal{F}_{\text{robust-standard-reshare}}$ sends the parties of committee C_{i+5}* abort*, then they abort.*

Proof. This is immediate from the definitions of $\mathcal{F}_{\text{robust-packed-reshare}}$ and $\mathcal{F}_{\text{robust-standard-reshare}}$. □

Theorem 3.27. *Protocol $\Pi_{\text{main-2/3-hm}}$ UC-securely computes $\mathcal{F}_{\text{DABB}}$ in the presence of an R-adaptive adversary $\mathcal{A}$ in the $(\mathcal{F}_{\text{input}}, \mathcal{F}_{\text{robust-packed-reshare}}, \mathcal{F}_{\text{robust-standard-reshare}}, \mathcal{F}_{\text{rand-2/3-hm}}, \mathcal{F}_{\text{output}})$-hybrid model.*

Proof. Assume w.l.o.g., that the first $2t+1$ parties are honest. First, we define simulator $\mathcal{S}$:

1. For each input x, $\mathcal{S}$ emulates $\mathcal{F}_{\text{input}}$ – in case the client is corrupted, $\mathcal{S}$ gets from the adversary the sharing $[x]_{2t}^{C_{\text{clnt}}}$, reconstructs x and sends it to $\mathcal{F}_{\text{DABB}}$; otherwise, $\mathcal{S}$ gets from the adversary the t corrupted clients' shares of $[x]_{2t}^{C_{\text{clnt}}}$.

2. During the execution phase, for each wire value x, the corresponding sharing $[x]_{2t}^{C_i}$ might have some additive error ε_x (for the first layer, each $\varepsilon_x = 0$). $\mathcal{S}$ will maintain the invariant of

computing the corrupted parties' shares of $[x + \varepsilon_x]_{2t}^{C_i}$ for every wire value x – as a base case, it already has these for all circuit inputs.

3. For identity gates:

   (a) $\mathcal{S}$ first emulates $\mathcal{F}_{\text{robust-packed-reshare}}$ – $\mathcal{S}$ sends to the adversary the corrupted parties' shares of $\left[x_1 + \varepsilon_{x_1}\right]_{2t}^{C_i}, \dots, \left[x_{t+1} + \varepsilon_{x_{t+1}}\right]_{2t}^{C_i}$ that it has already computed, then receives back a set of shares $\{\mathbf{x}^k\}_{k\in[2t+2,n]}$ and error vector $\mathbf{\Delta}$.

   (b) Then $\mathcal{S}$ emulates $\mathcal{F}_{\text{robust-standard-reshare}}$ – $\mathcal{S}$ sends to the adversary the just received shares $\{\mathbf{x}^k\}_{k\in[2t+2,n]}$ and receives back set of shares $\{(x_1^k, \dots, x_{t+1}^k)\}_{k\in[2t+2,n]}$ (thus it has all of the corrupted parties' shares of the gates' output wires) and errors $(\delta_1, \dots, \delta_{t+1})$.

   (c) $\mathcal{S}$ next computes the new error $\varepsilon'_{x_\alpha} \leftarrow \Delta_\alpha + \delta_\alpha$ on the output wire of the gate, for $\alpha \in [t+1]$.

   (d) Finally, if $\mathbf{\Delta} \neq \mathbf{0}$ then $\mathcal{S}$ aborts for committee C_{i+3} and if $(\delta_1, \dots, \delta_{t+1}) \neq (0, \dots, 0)$, then $\mathcal{S}$ aborts for committee C_{i+4}.

4. Addition gates proceed similarly.

5. For multiplication gates:

   (a) $\mathcal{S}$ first emulates $\mathcal{F}_{\text{rand-2/3-hm}}$ – it receives from the adversary the set of shares $\{(a_1^k, b_1^k, \dots, a_{t+1}^k, b_{t+1}^k)\}_{k\in[2t+2,n]}$ and sharings $([\eta_1]_{t'}^{C_i}, \dots, [\eta_{t+1}]_{t'}^{C_i})$.

   (b) Then, $\mathcal{S}$ samples random $a_1^*, \dots, a_{t+1}^*$ and for each $\alpha \in [t+1]$ s.t. x_α was not input by a corrupted party, $x_\alpha^1, \dots, x_\alpha^t$, and similarly random $b_1^*, \dots, b_{t+1}^*$ and for each $\alpha \in [t+1]$ s.t. y_α was not input by a corrupted party, $y_\alpha^1, \dots, y_\alpha^t$.

   (c) Based on these and the corrupted parties' shares of inputs $\left[x_1 + \varepsilon_{x_1}\right]_{2t}^{C_i}, \dots, \left[x_{t+1} + \varepsilon_{x_{t+1}}\right]_{2t}^{C_i}$ and $\left[y_1 + \varepsilon_{y_1}\right]_{2t}^{C_i}, \dots, \left[y_{t+1} + \varepsilon_{y_{t+1}}\right]_{2t}^{C_i}$ that $\mathcal{S}$ has already computed (or all such shares for those x_α, y_α input by corrupted parties), $\mathcal{S}$ computes the honest parties' shares of $\left[d'_k\right]_{2t}^{C_i}$ and $\left[e'_k\right]_{2t}^{C_i}$ for $k \in [2t+2, n]$, as well as values d'_k, e'_k for $k \in [2t+1]$, according to the proof of Lemma 3.24.

(d) $\mathcal{S}$ then sends to the adversary the honest parties' shares of $\left[d'_k\right]_{2t}^{C_i}$ and $\left[e'_k\right]_{2t}^{C_i}$ just computed.

(e) Then, $\mathcal{S}$ emulates $\mathcal{F}_{\text{robust-packed-reshare}}$ three independent times – $\mathcal{S}$ sends to the adversary the above received corrupted parties' shares $\{(a_1^k, \ldots, a_{t+1}^k)\}_{k\in[2t+2,n]}$ and $\{(b_1^k, \ldots, b_{t+1}^k)\}_{k\in[2t+2,n]}$, as well as $\{(a_1^k \cdot b_1^k, \ldots, a_{t+1}^k \cdot b_{t+1}^k)\}_{k\in[2t+2,n]}$.

(f) It receives back a set of shares $\{\mathbf{a}^k\}_{k\in[2t+2,n]}$, $\{\mathbf{b}^k\}_{k\in[2t+2,n]}$, and $\{\mathbf{c}^k\}_{k\in[2t+2,n]}$, as well as error vectors $\boldsymbol{\Delta}_\mathbf{a}$, $\boldsymbol{\Delta}_\mathbf{b}$, and $\boldsymbol{\Delta}_\mathbf{c}$.

(g) $\mathcal{S}$ then sends to the adversary the values d'_k, e'_k for $k \in [2t+1]$, computed above.

(h) Next, $\mathcal{S}$ emulates $\mathcal{F}_{\text{robust-standard-reshare}}$ three independent times – $\mathcal{S}$ sends to the adversary the just received shares $\{\mathbf{a}^k\}_{k\in[2t+2,n]}$, $\{\mathbf{b}^k\}_{k\in[2t+2,n]}$, and $\{\mathbf{c}^k\}_{k\in[2t+2,n]}$.

(i) It then receives back sets of shares $\{(a_1^k, \ldots, a_{t+1}^k)\}_{k\in[2t+2,n]}$, $\{(b_1^k, \ldots, b_{t+1}^k)\}_{k\in[2t+2,n]}$, and $\{(c_1^k, \ldots, c_{t+1}^k)\}_{k\in[2t+2,n]}$ as well as errors $(\delta_{a_1}, \ldots, \delta_{a_{t+1}})$, $(\delta_{b_1}, \ldots, \delta_{b_{t+1}})$, and $(\delta_{c_1}, \ldots, \delta_{c_{t+1}})$.

(j) Then $\mathcal{S}$ computes $\{d_\alpha \cdot e_\alpha - d_\alpha \cdot b_\alpha^k - e_\alpha \cdot a_\alpha^k + c_\alpha^k\}_{\alpha\in[t+1],k\in[2t+2,n]}$ as the corrupted parties' shares of the gates' output wires ($d_1, e_1, \ldots, d_{t+1}, e_{t+1}$ can be directly computed from $d'_1, e'_1, \ldots, d'_{t+1}, e'_{t+1}$).

(k) $\mathcal{S}$ next computes the new error $\varepsilon_{z_\alpha} \leftarrow d_\alpha \cdot (\Delta_{b_\alpha} + \delta_{b_\alpha}) - e_\alpha \cdot (\Delta_{a_\alpha} + \delta_{a_\alpha}) + \Delta_{c_\alpha} + \delta_{c_\alpha}$ on the output wire of the gate, for $\alpha \in [t+1]$.

(l) Finally, if any of $\boldsymbol{\Delta}_\mathbf{a}, \boldsymbol{\Delta}_\mathbf{v}, \boldsymbol{\Delta}_\mathbf{c}$ are not $\mathbf{0}$ then $\mathcal{S}$ aborts for committee C_{i+3} and if any of $(\delta_{a_1}, \ldots, \delta_{a_{t+1}})$, $(\delta_{b_1}, \ldots, \delta_{b_{t+1}})$, $(\delta_{c_1}, \ldots, \delta_{c_{t+1}})$ are not $(0, \ldots, 0)$, then $\mathcal{S}$ aborts for committee C_{i+4}.

6. Finally, for each output z, $\mathcal{S}$ emulates $\mathcal{F}_{\text{output}}$:

(a) If the client is corrupted, $\mathcal{S}$ receives z from $\mathcal{F}_{\text{DABB}}$, and using the above computed error ε_z, the corrupted parties' shares of $[z + \varepsilon_z]_{2t}^{C_\ell}$ that it has computed, and t more random values for the first t honest parties shares, reconstructs all of $[z + \varepsilon_z]_{2t}^{C_\ell}$ then sends it to the adversary.

(b) Otherwise, if the client is honest $\mathcal{S}$ sends the corrupted parties' shares of outputs $[z + \varepsilon_z]_{2t}^{C_\ell}$ that it has computed to the adversary.

Now we argue that the real world and ideal world are distributed identically to the adversary. It is clear that $\mathcal{S}$ correctly reconstructs the corrupted parties' inputs x in the ideal world. Therefore all outputs received from $\mathcal{F}_{\text{DABB}}$ by $\mathcal{S}$ will be consistent with the adversary's inputs. Additionally, it is clear that $\mathcal{S}$ has the corrupted parties' shares of the input sharings corresponding to the real world. We will show that $\mathcal{S}$ maintains the invariant of computing the corrupted parties' shares for every wire. Furthermore, it is clear that for all inputs that do not come from corrupted clients, the first t honest parties' shares are distributed randomly to the real world adversary. We will furthermore show that this invariant is maintained for all wires that are not corrupted clients' inputs.

For identity and addition gates, assuming the invariant, $\mathcal{S}$ clearly sends to the adversary the same corrupted parties' shares that it would receive in the real world. $\mathcal{S}$ additionally aborts when the parties in the real world would (according to Lemma 3.26). It is also clear that $\mathcal{S}$ maintains the invariant of computing the corrupted parties' shares for every wire. Furthermore, from the definition of $\mathcal{F}_{\text{robust-standard-reshare}}$ it is clear that the invariant that the first t honest parties' shares are distributed randomly to the real world adversary for every wire is maintained.

For multiplication gates, because of the invariant that the first t honest parties' shares are distributed randomly to the real world adversary for every wire, we know that the assumption on which Lemma 3.24 is based is true. Therefore using the lemma, we know that $\mathcal{S}$ computes and sends to the adversary honest parties' shares of $\left[d'_k\right]_{2t}^{C_i}$ and $\left[e'_k\right]_{2t}^{C_i}$ for $k \in [2t+2, n]$, as well as values

d'_k, e'_k for $k \in [2t+1]$, that are distributed identically to those in the real world. It is also clear that $\mathcal{S}$ sends to the adversary the same corrupted parties' shares when emulating $\mathcal{F}_{\text{robust-packed-reshare}}$ and $\mathcal{F}_{\text{robust-standard-reshare}}$ that it would receive in the real world. $\mathcal{S}$ additionally aborts when the parties in the real world would (according to Lemma 3.26). Finally, $\mathcal{S}$ also maintains the invariant of computing the corrupted parties' shares for every wire, as it can compute $d_1, e_1, \ldots, d_{t+1}, e_{t+1}$ directly and receives from the adversary the corrupted parties' shares of the reshared multiplication triples. Furthermore, from the definition of $\mathcal{F}_{\text{robust-standard-reshare}}$, it is clear that the first t honest parties' shares of the multiplication triple are distributed randomly to the real world adversary, and thus the invariant that the first t honest parties' shares are distributed randomly to the real world adversary for every wire is maintained.

Thus the execution phase is simulated perfectly and all that remains to show is that the output phase is simulated perfectly. First, we recall that from Lemma 3.25 that during every run of π_{mult} of the execution phase, the parties of committee C_{i+2} always correctly determine the values of $(x_1 + \varepsilon_{x_1} + a_1, y_1 + \varepsilon_{y_1} + b_1, \ldots, x_{t+1} + \varepsilon_{x_{t+1}} + a_{t+1}, y_{t+1} + \varepsilon_{y_{t+1}} + b_{t+1})$. Now, for any circuit layer starting with committee C_i, observe that if the adversary injects any error via $\mathcal{F}_{\text{robust-packed-reshare}}$ or $\mathcal{F}_{\text{robust-standard-reshare}}$, then in the real world, at the latest, the honest parties of committee C_{i+4} will abort. Since for the next layer, the shares of the output wires of the gates are not computed until C_{i+4}, that means that if the parties do not abort at the end of computing a layer, then the errors on the output wires from $\mathcal{F}_{\text{robust-packed-reshare}}$ and $\mathcal{F}_{\text{robust-standard-reshare}}$ can only come from the computation of that layer, and not any previous layers. Similarly, if the adversary injects any error via $\mathcal{F}_{\text{rand-2/3-hm}}$, then in the real world, at the latest, the honest parties of committee C_{i+2} will abort, which means that the errors on the output wires for any layer can only come from $\mathcal{F}_{\text{robust-packed-reshare}}$ and $\mathcal{F}_{\text{robust-standard-reshare}}$ for that layer. Therefore, $\mathcal{S}$ properly computes the errors on the wires of the output gates. Based on these, and since the invariant that the first t honest parties' shares are distributed randomly to the real world adversary for every wire holds, it is clear that $\mathcal{S}$ simulates the output phase perfectly. Indeed, when emulating $\mathcal{F}_{\text{output}}$, $\mathcal{S}$ uses the same underlying secret $z + \varepsilon_z$ as in the real world, the corrupted parties' t shares and t random values for the first t honest parties shares to reconstruct the sharing $[z + \varepsilon_z]_{2t}^{C_\ell}$ to the adversary. This concludes the proof. $\square$

4 | Batch PIR and Labeled PSI with Oblivious Ciphertext Compression

In this chapter, we study a central component of Private Join and Compute protocols. In Private Join and Compute, two parties, each holding a different set of items with associated values, privately compute a function of the associated values of items in the intersection. As described in the introduction, the most computation- and communication-intensive part of such protocols is to privately compute the intersection of the sets. We improve the efficiency, in particular the communication complexity, of state-of-the art protocols for computing the intersection by defining and constructing *oblivious ciphertext compression* schemes. This chapter is based (often verbatim) on [Bienstock et al. 2024].

4.1 Preliminaries

4.1.1 Homomorphic Encryption

Throughout this chapter, we will define ciphertexts using $\tilde{c}$. A vector of ciphertexts will be defined as $\tilde{\mathbf{c}} = (\tilde{c}_1, \ldots, \tilde{c}_n)$.

In our work, we will mainly consider lattice-based somewhat homomorphic encryption (SHE) where parameters are chosen to support a limited number of homomorphic operations, as used in prior state-of-the-art constructions of batch PIR and labeled PSI [Mughees et al. 2021; Menon and

Wu 2022; Chen et al. 2018, 2017; Cong et al. 2021]. Our compression protocols only use additive hommorphism of these schemes, where noise grows additively.

In this section, we outline two classes of SHE schemes: one with small ciphertext expansion but large noise growth and the other with large ciphertext expansion and small noise growth. The first class (including Regev [Regev 2005] and BFV [Brakerski 2012; Fan and Vercauteren 2012]) are SHE schemes with small ciphertext expansion (the ratio of ciphertext size to plaintext size), but large noise growth especially for ciphertext-ciphertext multiplication. In contrast, the second class of schemes (including GSW [Gentry et al. 2013]) are SHE schemes with large ciphertext expansion, but very small noise growth for ciphertext-ciphertext expansion. In particular, GSW [Gentry et al. 2013] ensures only additive noise growth whereas the first class of SHE schemes require multiplicative noise growth when performing homomorphic multiplications. Finally, it is shown that protocols can perform operations using ciphertexts from different classes. Recent PIR schemes [Mughees et al. 2021; Menon and Wu 2022] rely on multiplying ciphertexts from each class resulting in small noise growth.

Regev and BFV Encryption. Many PIR schemes relied upon Regev encryption [Regev 2005] equipped with homomorphic addition and its extension by Brakerski [Brakerski 2012] as well as Fan and Vercauteren [Fan and Vercauteren 2012] enabling homomorphic multiplication. These schemes are defined over a ring $\mathbb{R} = \mathbb{Z}/(x^n + 1)$ where n is the dimension of the polynomial along with a plaintext and ciphertext modulus q and t respectively. A plaintext value is a polynomial in R $\bmod t$ and a ciphertext consists of $\tilde{c} = (c_0, c_1)$ where both polynomials are elements of $R \mod q$. For more information on the details of these schemes, we defer readers to prior works [Regev 2005; Brakerski 2012; Fan and Vercauteren 2012]. We will only use them in a blackbox manner with certain properties that we describe next.

First, we describe the noise growth of each homomorphic operation. For ciphertext-ciphertext addition, we note that noise growth is additive. In particular, if we consider two ciphertexts $\tilde{c}_1$ and

$\tilde{c}_2$ with error $\mathsf{Err}(\tilde{c}_1)$ and $\mathsf{Err}(\tilde{c}_2)$, then the resulting error is $O(\mathsf{Err}(\tilde{c}_1) + \mathsf{Err}(\tilde{c}_2))$ after homomorphic addition. For ciphertext-plaintext multiplication (also known as absorption) with a ciphertext $\tilde{c}$ with error $\mathsf{Err}(\tilde{c})$ and any plaintext message m, the resulting error is $O(|m| \cdot \mathsf{Err}(\tilde{c}))$. Finally, for ciphertext-ciphertext multiplication, the noise growth becomes $O(t \cdot (\mathsf{Err}(\tilde{c}_1) + \mathsf{Err}(\tilde{c}_2)))$. Note, for a sequence of ciphertext-ciphertext multiplications, the noise growth would grow exponentially in the length of the sequence. As a result, recent PIR schemes avoid these operations.

Finally, these schemes have been shown to emit properties that enable packing multiple plaintext values into a single ciphertext that has been used to reduce PIR request sizes [Angel et al. 2018].

GSW Encryption. The second class of SHE scheme is the Gentry, Sahai and Waters scheme [Gentry et al. 2013] that can be defined over the same polynomial ring $\mathbb{R} = \mathbb{Z}/(x^n + 1)$, plaintext space $\mathbb{R} \mod t$ and ciphertext space $\mathbb{R} \mod q$. The scheme is parameterized by a base B and length ℓ that provides a trade-off between noise growth and efficiency. Once again, we defer details of the GSW scheme to prior works [Gentry et al. 2013]. Instead, we will only provide details about certain properties that will be leveraged in our work.

In particular, we will rely on the external product operation introduced by [Chillotti et al. 2020]. The input to the external product is a Regev/BFV ciphertext and a GSW ciphertext and the output is a Regev/BFV ciphertext containing the multiplication of the two input ciphertexts. The main advantage of the external product is that noise growth is linear and asymmetric. For a Regev/BFV ciphertext $\tilde{c}_1$ with error $\mathsf{Err}(\tilde{c}_1)$ and a GSW ciphertext $\tilde{c}_2$ with error $\mathsf{Err}(\tilde{c}_2)$, the output of the external product is a Regev/BFV ciphertext with noise $O(B \cdot \mathsf{Err}(\tilde{c}_2) + \mathsf{Err}(\tilde{c}_1))$ requiring $O(\ell)$ polynomial multiplications. As earlier stated, the choice of B and ℓ provide trade-offs between the noise growth and efficiency of the external product operation. Secondly, we note that the noise growth is asymmetric in the sense that noise grows linearly in the Regev/BFV ciphertext and the B multiplicative factor only affects the GSW ciphertext.

4.1.2 Oblivious Ciphertext Compression

We define the notion of an *oblivious ciphertext compression* scheme. For this primitive, we only assume additive homomorphism (ciphertext-ciphertext addition). The problem consists of two parties: a compressor and a decompressor. The compressor is given n ciphertexts, $\tilde{\mathbf{c}} = (\tilde{c}_1, \ldots, \tilde{c}_n)$, to be compressed. Both the compressor and the decompressor know the number of non-zero plaintext entries t. In addition, the decompressor has the private decryption key and the indices of the t non-zero entries, $I \subset [n]$. If $i \in I$, then $\tilde{c}_i$ is an encryption of a non-zero entry. The compressor's job is to produce a succinct encoding of the input ciphertexts with knowledge of only t. The encoding is consumed by the decompressor to recover the original t non-zero plaintext entries. We formally define oblivious ciphertext compression below.

Definition 4.1 (Oblivious Ciphertext Compression)**.** Let $\mathbf{p} = (p_1, \ldots, p_n) \in \mathbb{F}^n$ be a vector of n plaintexts with at most t non-zero entries. Let $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ be an additive homomorphic encryption scheme, and let $\tilde{\mathbf{c}} = (\tilde{c}_1, \ldots, \tilde{c}_n)$ where $\tilde{c}_i = \mathcal{E}.\mathsf{Enc}(\mathbf{pk}_{\mathcal{E}}, p_i)$ for each $i \in [n]$. An oblivious ciphertext compression scheme consists of a pair of algorithms ObvCompress and Decompress satisfying:

- $\hat{\mathbf{c}} \leftarrow \mathsf{ObvCompress}(\mathbf{pk}_{\mathcal{E}}, \tilde{\mathbf{c}}, t; R)$: Oblivious compression takes in a public key $\mathbf{pk}_{\mathcal{E}}$, n ciphertexts $\tilde{\mathbf{c}} = [\tilde{c}_1, \ldots, \tilde{c}_n]$, the number of non-zero plaintext entries t, and randomness R. It outputs compressed ciphertexts $\hat{\mathbf{c}}$.

- $\mathbf{p} \leftarrow \mathsf{Decompress}(\mathbf{sk}_{\mathcal{E}}, \hat{\mathbf{c}}, I; R)$: Decompression takes in a secret key $\mathbf{sk}_{\mathcal{E}}$, compressed ciphertexts $\hat{\mathbf{c}}$, the non-zero plaintext entry indices $I \subset [n]$ ($|I| \leqslant t$) of p, and randomness R. It outputs the non-zero plaintext values $\{i, p_i\}_{i \in I}$.

Let $\gamma = \gamma(\lambda)$ be the bit length of all n ciphertexts produced by the homomorphic encryption scheme $\mathcal{E}$. An oblivious ciphertext compression is δ-compressing if the bit length of $\hat{\mathbf{c}}$ is at most

$\delta \cdot \gamma \cdot |\tilde{\mathbf{c}}|$. The failure probability is at most ϵ if, for each plaintext vector $\mathbf{p} = (p_1, ..., p_n)$ and associated ciphertexts $\tilde{\mathbf{c}} = (\tilde{c}_1, \ldots, \tilde{c}_n)$ with at most t non-zero values,

$$\Pr[\mathsf{Decompress}(\mathbf{sk}_{\mathcal{E}}, \hat{\mathbf{c}}, I) \neq \{i, p_i\}_{i \in I}] \leqslant \epsilon$$

where $\hat{\mathbf{c}} \leftarrow_{\$} \mathsf{ObvCompress}(\mathbf{pk}_{\mathcal{E}}, \tilde{\mathbf{c}}, t)$.

Comparison with Prior Work. Liu and Tromer [Liu and Tromer 2022] implicitly studied oblivious ciphertext compression, without explicitly defining the primitive. Fleischhacker *et al.* [Fleischhacker et al. 2023] considered another variant closer to our compression problem that was also implicitly studied in [Liu and Tromer 2022]. where the decompressor is not given the identity of the non-zero plaintext indices, $I \subset [n]$. Therefore, this is a harder setting than our compression problem. It is not surprising that the resulting compression rates or decoding efficiency are significantly worse than our constructions (see Figure 1.1). To our knowledge, our specific variant of compression has not been explicitly studied previously.

4.1.3 Oblivious Ciphertext Decompression

Next, we define *oblivious ciphertext decompression* that switches the compressor and decompressor roles. The compressor is given the plaintext vector, $\mathbf{p} = (p_1, \ldots, p_n)$ and a subset of t indices, $I \subset [n]$ with $|I| = t$ to produce a succinct encoding $\hat{\mathbf{c}}$. The decompressor is given $\hat{\mathbf{c}}$ and must produce the ciphertext vector $\tilde{\mathbf{c}} = (\tilde{c}_1, \ldots, \tilde{c}_n)^T$ such that each $\tilde{c}_i$ is an encryption of p_i for all $i \in I$. No correctness is required for $i \notin I$. In other words, $\tilde{c}_i$ needs to be an encryption of p_i only when $i \in I$. However, the decompressor must obliviously decode without any knowledge of the relevant indices, I. In fact, the compressed ciphertexts $\hat{\mathbf{c}}$ must not reveal any information about neither the underlying plaintext values $\mathbf{p} = (p_1, \ldots, p_n)^T$ nor the relevant indices I. To our knowledge, no prior works have studied this setting.

Definition 4.2 (Oblivious Ciphertext Decompression)**.** Let $p = (p_1, ..., p_n)^T \in \mathbb{F}^n$ be a vector of n plaintexts and $I \subset [n]$ be a subset of $t < n$ indices. Let $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ be an additive homomorphic encryption scheme. A oblivious ciphertext decompression scheme consists of a pair of algorithms (Compress, ObvDecompress), where:

- $\hat{\mathbf{c}} \leftarrow \mathsf{Compress}(\mathbf{sk}_{\mathcal{E}}, p, I; R)$: The compression algorithm takes in a secret homomorphic encryption key $\mathbf{sk}_{\mathcal{E}}$, a vector of n plaintexts $p = (p_1, ..., p_n)^T$, a subset of t indices $I \subset [n]$ and randomness R. Then, it outputs the compressed ciphertexts $\hat{\mathbf{c}}$.
- $\mathbf{p} \leftarrow \mathsf{ObvDecompress}(\mathbf{pk}_{\mathcal{E}}, \hat{\mathbf{c}}, n; R)$: The decompression algorithm takes in a public homomorphic encryption key $\mathbf{pk}_{\mathcal{E}}$, compressed ciphertexts $\hat{\mathbf{c}}$, the number of total plaintexts n, and randomness R. Then, it outputs the ciphertext vector $\tilde{c} = (\tilde{c}_1, \ldots, \tilde{c}_n)^T$.

Let $\gamma = \gamma(\lambda)$ be the bit length of all n ciphertexts produced by the homomorphic encryption scheme $\mathcal{E}$. A oblivious ciphertext decompression is δ-compressing if the bit length of $\hat{\mathbf{c}}$ is at most $\delta \cdot \gamma \cdot |\tilde{c}|$. The failure probability is at most ϵ if, for each plaintext vector $p = (p_1, ..., p_n)^T$ and subset $I \subset [n]$ of size t, the following holds:

$$\Pr[\exists i \in I \mid \mathsf{Dec}(\mathbf{sk}_{\mathcal{E}}, \tilde{c}_i) \neq p_i] \leqslant \epsilon$$

where $\hat{\mathbf{c}} \leftarrow_{\$} \mathsf{Compress}(\mathbf{sk}_{\mathcal{E}}, p, I)$ and $(\tilde{c}_1, \ldots, \tilde{c}_n)^T \leftarrow_{\$} \mathsf{Decompress}(\mathbf{pk}_{\mathcal{E}}, \hat{\mathbf{c}})$. We note that there are no correctness requirements for ciphertexts $\tilde{c}_i$ such that $i \notin I$.

The scheme is computationally oblivious if, for all pairs of plaintext vectors $p = (p_1, \ldots, p_n)^T$ and $p' = (p'_1, \ldots, p'_n)^T$ and pairs of index sets $I, I' \subset [n]$ of size t, a computationally adversary cannot distinguish between the following:

- $\hat{\mathbf{c}} \leftarrow_{\$} \mathsf{Compress}(\mathbf{sk}_{\mathcal{E}}, p, I)$
- $\hat{\mathbf{c}}' \leftarrow_{\$} \mathsf{Compress}(\mathbf{sk}_{\mathcal{E}}, p', I')$.

```
$\mathbf{Expt}^{\mathsf{clnt}}_{b,\mathcal{A}}(\lambda, D)$

  1. $(\mathsf{prms}, \mathsf{ck}, \mathsf{sk}, E) \leftarrow_\$ \mathsf{BatchKWPIR.init}(1^\lambda, D)$
  2. $R \leftarrow \emptyset$
  3. For $i = 1, 2, \ldots, \mathsf{poly}(\lambda, D)$:
     (a) $(Q_0, Q_1) \leftarrow_\$ \mathcal{A}(\mathsf{prms}, \mathsf{sk}, E, R)$
     (b) $(\cdot, \mathsf{req}) \leftarrow_\$ \mathsf{BatchKWPIR.query}(\mathsf{prms}, \mathsf{ck}, Q_b)$
     (c) $R \leftarrow \cup\{\mathsf{req}\}$
  4. Return $b' \leftarrow_\$ \mathcal{A}(\mathsf{prms}, \mathsf{sk}, E, R)$
```

Figure 4.1: Experiment for batch keyword PIR client query privacy

4.1.4 Batch PIR and Labeled PSI

Batch (Keyword) PIR. In batch keyword PIR, the client holds a batch of ℓ keys, $\{q_1, \ldots, q_\ell\}$, and the server holds a public database $D \in (\mathcal{K} \times \mathcal{V})^n$ of n key-value pairs with n distinct keys, $\{(k_1, v_1), \ldots, (k_n, v_n)\}$. The client wishes to retrieve the database entries $\{D[q_1], \ldots, D[q_\ell]\}$ from the server. For any $q \in \mathcal{K}$, $D[q]$ denotes the value associated with key q. If $q = k_i$, then $D[q] = v_i$. Otherwise, $D[q] = \perp$. The following two properties must hold:

- *Correctness*: If the protocol is executed correctly, the client recovers $\{D[q_1], \ldots, D[q_\ell]\}$ as desired.
- *Query Privacy*: The server learns no information about the batch query, $\{q_1, \ldots, q_\ell\}$.

One can obtain the definition of single-query PIR if the batch query contains only a single index, $\ell = 1$. Furthermore, one can obtain non-keyword PIR if we restrict the database's key universe to be $\mathcal{K} = [n]$. Throughout this chapter, we will consider keyword PIR unless otherwise specified. We now provide a formal definition:

Definition 4.3 (Batch Keyword PIR)**.** A batch keyword private information retrieval (PIR) scheme over key universe $\mathcal{K}$, value universe $\mathcal{V}$, and batches of $\ell \geq 1$ quries consists of algorithms (BatchKWPIR.init, BatchKWPIR.query, BatchKWPIR.answer, BatchKWPIR.decrypt) satisfying:

- $(\mathsf{prms}, \mathsf{ck}, \mathsf{sk}, E) \leftarrow_{\$} \mathsf{BatchKWPIR.init}(1^\lambda, D)$: The initialization algorithm receives the security parameter λ and the database $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$. The output are the public parameters prms, a client key ck, a server key sk, and an encoding of the database E.

- $(\mathsf{st}, \mathsf{req}) \leftarrow_{\$} \mathsf{BatchKWPIR.query}(\mathsf{prms}, \mathsf{ck}, Q)$: The query algorithm receives the public parameters prms, client key ck, and a query set $Q = \{q_1, \ldots, q_\ell\} \in \mathcal{K}^\ell$ and outputs a state st and the request req to be sent to the server.

- $\mathsf{resp} \leftarrow \mathsf{BatchKWPIR.answer}(\mathsf{prms}, \mathsf{sk}, E, \mathsf{req})$: The answer algorithm receives hte public parameters prms, server key sk, encoding of the database E, and the request req and outputs a response resp.

- $(v_1, \ldots, v_\ell) \leftarrow \mathsf{BatchKWPIR.decrypt}(\mathsf{prms}, \mathsf{ck}, \mathsf{st}, \mathsf{resp})$: The process algorithm receives the public parameters prms, client key ck, state st, and response resp and outputs the values associated to the queried keys Q.

Additionally, the following guarantees must be satisfied:

- **Correctness**: For all databases D (with unique keys) and for all batch queries $Q \in \mathcal{K}^\ell$, the probability that the client's output when querying Q is not $\{D[q]\}_{q \in Q}$ is at most $\mathsf{negl}(\lambda)$.

- **Client Query Privacy**: We define client query privacy with respect to the experiment defined in Figure 4.1 parameterized by a bit $b \in \{0, 1\}$ and an adversary $\mathcal{A}$. For all databases D and all PPT adversaries $\mathcal{A}$, $\Pr_b[\mathbf{Expt}^{\mathsf{clnt}}_{b,\mathcal{A}}(\lambda, D) = b] \leq 1/2 + \mathsf{negl}(\lambda)$.

(Unbalanced) Labeled PSI. In labeled PSI, the receiver and sender hold sets X and Y respectively. The sender also holds a database of associated labels $\{L_y \mid y \in Y\}$. The goal is for the receiver to receive labels that appear in the intersection, $\{(z, L_z) \mid z \in X \cap Y\}$. The following properties must hold:

- *Correctness*: If the protocol is executed correctly, the receiver recovers $\{(z, L_z) \mid z \in X \cap Y\}$ as desired.
- *Receiver (Query) Privacy*: The sender learns no information about the receivers's set X beyond its size $|X|$.
- *Sender (Database) Privacy*: The receiver learns no information about the sender's set Y except for the desired output and its size $|Y|$.

In the unbalanced setting, the receiver's set X is typically much smaller than the sender's set Y, $|X| \ll |Y|$. Note, labeled PSI is similar to batch keyword PIR with the main difference being the additional sender (database) privacy guarantee. We now present the formal functionality of unbalanced labeled private set intersection (PSI), $\mathcal{F}_{\text{ul-psi}}$.

Figure 4.2: Functionality $\mathcal{F}_{\text{ul-psi}}$

Parameters: There are two parties, a receiver and a sender. The honest receiver and sender have respective set sizes $n_{\mathrm{X}}, n_{\mathrm{Y}}$. If the receiver or sender is maliciously corrupt, then their set size is n'_{X} or n'_{Y}, respectively.

Functionality:

1. On input $(\textsc{Receive}, \mathsf{sid}, X)$ from the receiver where $X \subseteq \{0,1\}^*$, ensure that $|X| \le n_{\mathrm{X}}$ if the receiver is honest and $|X| \le n'_{\mathrm{X}}$ otherwise. Give $(\textsc{Receiver-Input}, \mathsf{sid})$ to the sender.
2. Thereafter, on input $(\textsc{Send}, \mathsf{sid}, (Y, \{L_y \in \{0,1\}^{\ell} \mid y \in Y\}))$ from the sender where $Y \subseteq \{0,1\}^*$, ensure that $|Y| \le n_{\mathrm{Y}}$ if the sender is honest and $|Y| \le n'_{\mathrm{Y}}$, otherwise. Give output $(\textsc{Output}, \mathsf{sid}, \{(x, L_x) \mid x \in X \cap Y\}$ to the receiver.

4.2 Oblivious Ciphertext Compression

In this section, we present our oblivious ciphertext compression scheme, LSObvCompress, based on linear systems. We start with a simpler scheme before presenting our main construction.

4.2.1 First Attempt: Balls-into-Bins

In this section, we start with a construction which leverages the balls-into-bins random process. Given m bins and n balls, each of the n balls are thrown into one of the m bins uniformly at random. In the context of ciphertext compression, bins correspond to compressed ciphertexts and balls correspond to input non-zero ciphertexts. Throwing a ball into a bin corresponds to homomorphically adding an input ciphertext to one of the compressed ciphertexts. Decompression works by re-simuluating the ball throws for non-zero ciphertexts and decrypting the values at relevant bins. The main observation is that adding a zero-encrypting ciphertext can be thought of as "skipping" the ball throw, as its addition doesn't change the value of the underlying plaintext. Conceptually, the algorithm fails if any of the bins contains more than one ball. We describe the algorithm below.

We suppose that both parties share a hash function H. Upon receiving the input ciphertexts $\tilde{\mathbf{c}} = (\tilde{c}_1, \ldots, \tilde{c}_n)^T$ and the number of non-zero plaintext entries t, the compression algorithm first initializes a vector of $m \geq t$ zero ciphertexts $\hat{\mathbf{c}} = (\hat{c}_1, ..., \hat{c}_m)^T$, where $\hat{c}_i \leftarrow_\$ \mathcal{E}.\mathsf{Enc}(\mathbf{pk}_\mathcal{E}, 0)$. Then, for each input ciphertext $\tilde{c}_i$, the algorithm executes the following two operations. First, compute index $j = H(i) \in [m]$ where H is a random function with range $[m]$. Next, homomorphically add $\tilde{c}_i$ to $\hat{c}_j$, that is, $\hat{c}_j \leftarrow \mathcal{E}.\mathsf{Eval}(\mathbf{pk}_\mathcal{E}, +, (\tilde{c}_i, \hat{c}_j))$. Finally, the algorithm outputs the resulting vector $\hat{\mathbf{c}}$.

The decompression algorithm receives the compression $\hat{\mathbf{c}} = (\hat{c}_1, \ldots, \hat{c}_m)^T$ and non-zero plaintext entry indices I. For every non-zero ciphertext index $i \in I$, the algorithm computes $j = H(i)$ and sets $p_i \leftarrow \mathcal{E}.\mathsf{Dec}(\mathbf{sk}_\mathcal{E}, \hat{c}_j)$. Finally, the algorithm outputs all non-zero plaintext values, $\{i, p_i\}_{i \in I}$.

Note this algorithm can recover the original plaintext vector as long as the hash outputs $H(i)$

are all distinct for every $i \in I$. However, the probability of collision is high unless $m = \Omega(t^2)$ (due to the birthday problem) that is a quadratic blowup with respect to t. Ideally, we would like m to be not much larger than t to obtain an efficient compression rate.

Reformulating as a Linear System. We generalize the aforementioned scheme as constructing and solving a system of linear equations. More specifically, the compression algorithm is responsible for constructing a linear system that the decompression algorithm attempts to solve to recover the original plaintext vector. While this viewpoint seems rather unnecessarily complex, it will serve as an important basis to our main construction. We outline the reformulated algorithm below.

For each $i \in [n]$, the compression algorithm constructs a column vector $\mathbf{v}_i \in \mathbb{F}^m$ where only the $H(i)$-th element is set to 1 and the rest are set to 0. Let $\mathbf{M} = (\mathbf{v}_1, ..., \mathbf{v}_n) \in \mathbb{F}^{m\times n}$ be a matrix. Note that both parties know matrix $\mathbf{M}$ as they share hash function H. The compression algorithm computes and outputs the matrix-vector multiplication $\hat{\mathbf{c}} \leftarrow \mathbf{M} \cdot \tilde{\mathbf{c}}$.

The decompression algorithm takes in the vector $\hat{\mathbf{c}}$ and produces its decryption $\hat{\mathbf{p}}$. Next, we reconstruct the matrix $\mathbf{M}$ using the random function H. Let $I = \{i_1, ..., i_t\}$ be the set of non-zero plaintext entry indices, and let $\mathbf{M}_{\mathrm{c}(I)} = (\mathbf{v}_{i_1}, ..., \mathbf{v}_{i_t}) \in \mathbb{F}^{m\times t}$ be a sub-matrix of $\mathbf{M}$ consisting of all column vectors whose indices appear in I. Similarly, let $\hat{\mathbf{p}}_I = (\hat{p}_{i_1}, \ldots, \hat{p}_{i_t})$ for entries of $\hat{\mathbf{p}}$ in I. The algorithm solves the linear system associated with $\mathbf{M}_{\mathrm{c}(I)}$ and $\hat{\mathbf{p}}$ to compute $\mathbf{p}_I$ satisfying $\mathbf{M}_{\mathrm{c}(I)} \cdot \mathbf{p}_I = \hat{\mathbf{p}}_I$ to recover the non-zero $p_{i_j} = (\mathbf{p}_I)_j$ for each $j \in [t]$.

We note that the decompression algorithm can correctly recover the plaintext vector if and only if the linear system $\mathbf{M}_{\mathrm{c}(I)} \cdot \mathbf{p}_I = \hat{\mathbf{p}}_I$ has a unique solution (that is, $\mathbf{M}_{\mathrm{c}(I)}$ has full column rank). For our choice of $\mathbf{M}$, this precisely happens when all hash outputs $H(i)$ are distinct for every $i \in I$.

4.2.2 Second Attempt: Random Matrices

Recall that in the first attempt, the generated matrix $\mathbf{M}$ consists of random column vectors with Hamming weight exactly one corresponding to the balls-into-bins process. This forced us to set the number of rows and the encoding size to $m = \Omega(t^2)$ to avoid collisions. Taking a closer look, we notice that the way we generate the column vectors are unnecessarily restrictive. Indeed, for our scheme to succeed, we only require the $\mathbf{M}_{\mathsf{c}(I)}$ to have a unique solution. There is no need to restrict rows to Hamming weight one vectors.

This crucial observation leads to the following approach. Instead of sampling random column vectors with Hamming weight 1, we instead sample column vectors uniformly at random from $\{0,1\}^m$. To do this, we can imagine the shared hash function $H : [n] \to \{0,1\}^m$ outputs random binary column vectors of length m. Then, the shared matrix is $\mathbf{M} = (H(1), \ldots, H(n))$. This way, the generated column vectors will be linearly independent with high probability even when m is small. The rest of the algorithm stays identical.

Failure Probability and Compression Rate. The algorithm's failure probability and compression rate will be parameterized by ϵ and t. Let $m = (1+\epsilon)t$ be the number of rows. Even when ϵ is very small, the generated $m \times t$ matrix $\mathbf{M}_{\mathsf{c}(I)}$ has a unique solution except with negligible probability. For example, setting $m = t + \lambda$ with very small $\epsilon = \lambda/t$, the system has full rank with probability $1 - 2^{-\lambda-1}$ (see [Garimella et al. 2021]). The compression rate is almost optimal as the encoding contains $t + \lambda$ ciphertexts that is only λ more than the optimal minimum.

Running Time. Let $m = (1+\epsilon)t$. We start by analyzing the compression time. Generating a random column vector $\in \{0,1\}^m$ takes $O(m)$ time, so the entire matrix generation takes $O(mn)$ time during compression. Computing the matrix-vector product takes $m \cdot n$ homomorphic ciphertext additions. The compression algorithm performs $O(m \cdot t)$ ciphertext-ciphertext additions. For decompression, we note that solving the linear system associated to $\mathbf{M}_{\mathsf{c}(I)}$ requires $O(m \cdot t^2)$ time using Gaussian elimination.

Comparison to the First Attempt. While the new algorithm can give us very high compression rate, it is computationally very inefficient. Compression requires $O(mt)$ time and decompression requires $O(mt^2)$ time using Gaussian elimination. In practice, this may not be so problematic when $t << n$, but as t grows, the scheme is computationally expensive. Ideally, we would like compression to be close to linear in the number of ciphertexts, n, and decompression to be close to linear in t. In contrast, the first attempt has horrible compression rate of $m = O(t^2)$, but is computationally more efficient. The compression algorithm requires only $O(n)$ time. Furthermore, decompression only used $O(t^2)$ time.

This raises the following question: is it possible to get the best of both worlds - an algorithm that achieves high compression rate but is also practically efficient? We show that this is possible in the next subsection.

4.2.3 LSObvCompress: Random Band Matrices

In prior attempts, we generated random matrices uniformly at random from $\{0, 1\}^{m \times n}$. This allowed the associated random linear systems to be uniquely solvable with high probability even when $m = (1 + \epsilon)t$ was very small. However, solving this linear system is very inefficient, which made the previous scheme impractical for larger t. This is not too surprising, because the generated matrix is very dense. The expected number of non-zero matrix entries is $mn/2$. This suggests that the algorithm for solving the linear system must also have at least $O(mn)$ running time as well.

Looking closely, we again realize that we never needed the generated matrices to be sampled uniformly at random from $\{0, 1\}^{m \times n}$. That is, as long as the associated linear system is uniquely solvable with high probability, the distribution itself is irrelevant to the security of the scheme. Therefore, we only require a matrix generation algorithm that generates a "small" linear system that is uniquely and efficiently solvable. For LSObvCompress, we consider random matrices that

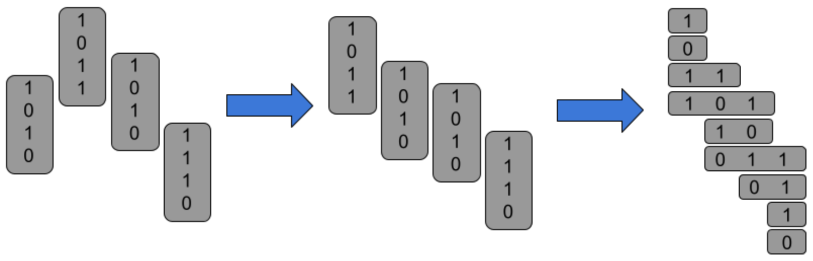

Figure 4.3: Example of a random band column matrix construction with band width $w = 4$. Second diagram shows the matrix after sorting the columns by the band start positions. Third diagram shows the random band row matrix view of the constructed matrix. In this example, the maximum band row width is 3.

satisfy these two properties.

Random Band Matrices. There has been extensive research on the core algorithmic problem of generating sparse random matrices that are efficiently solvable. For LSObvCompress, we utilize the random band matrices of Dietzfelbinger and Walzer [Dietzfelbinger and Walzer 2019] that is the most efficient to our knowledge.

Random band matrices are constructed such that each row consists of a random band with width w, and all entries outside of the band are zero. Formally, let m be the length of each row of the matrix. For each row, a band start index s is chosen randomly from $[m - w + 1]$, and each entry within the band, i.e. in range $[s, s + w)$, is a uniformly random bit from $\{0, 1\}$. All other entries outside the range $[s, s + w)$ remain 0.

Intuitively, random band matrices are solvable in $O(nw)$ time because the generated random matrix is "almost diagonal" after the rows are sorted by the band start positions. Furthermore, each row reduction operation maintains an invariant where the number of non-zero entries per rows is $O(w)$ making Gaussian elimination very efficient.

Adaptation for LSObvCompress. Unfortunately, we are unable to directly apply random band matrices for LSObvCompress. Going back to the linear system framework presented in Section 4.2.1, the client will solve the linear system associated with the matrix $\mathbf{M}_{c(I)}$. Recall that I

is the subset of non-zero plaintexts, $\mathbf{M}$ is the chosen random matrix and $\mathbf{M}_{c(I)}$ is the sub-matrix of $\mathbf{M}$ consisting of all the column vectors whose indices appear in I. Suppose we chose $\mathbf{M}$ to be a random band matrix. Unfortunately, $\mathbf{M}_{c(I)}$ is not guaranteed to be a random band matrix. In particular, it is possible that I (and, thus, the columns) are chosen such that each matrix row will have a band much smaller than length w or be all zero. In this case, it is unclear if the matrix $\mathbf{M}_{c(I)}$ still has a unique solution.

Instead, we will choose our matrix $\mathbf{M}$ using an adaptation of random band matrices to ensure that $\mathbf{M}_{c(I)}$ is still efficiently solvable for any choice of non-zero plaintext indices I. To do this, we will instead choose $\mathbf{M}$ to be the tranpose of random band matrices. In other words, we will generate each column vector of $\mathbf{M}$ to consist of a random band of width w. To do this, we imagine both parties share two hash functions $\mathsf{H}_1 : [n] \rightarrow [m - w + 1]$ and $\mathsf{H}_2 : [n] \rightarrow \{0, 1\}^w$. For the i-th column of the shared matrix $\mathbf{M}$, $\mathsf{H}_1(i)$ denotes the start of the band and $\mathsf{H}_2(i)$ chooses the random w-bit band.

Next, we can consider any subset of non-zero plaintexts I and the associated sub-matrix $\mathbf{M}_{c(I)}$. As each column vector consists of a random w-length band, $\mathbf{M}_{c(I)}$ remains a transpose of a random band matrix. As the column and row rank of any matrix is identical, we can rely on the analysis of Dietzfelbinger and Walzer [Dietzfelbinger and Walzer 2019] to see that $\mathbf{M}_{c(I)}$ will have a unique solution with high probability for any choice of I.

The only caveat is that we cannot apply the running time analysis of the random band row matrix construction, as the bands are constructed column-wise instead of row-wise. Nonetheless, we show that solving the system remains practically efficient with this modification in our experiments (see Section 4.6.1). Intuitively, this is because a transpose of a random band row matrix remains similar to a random band row matrix after the columns are sorted by the band start positions. The maximum band width across the entire rows is not much larger than the column band width w, which allows the linear system to be solved efficiently just as in the random band row matrix construction. See Figure 4.3 for an illustration.

More formally, we show that each row will consist of exactly one continguous section of non-zero entries of length $O(w)$. We couple the process of generating random band matrices with random column vectors as two-dimensional balls-into-bins allocation (see [Asharov et al. 2016] for more details). In particular, we model each of the t columns as t lists of w items. There exists $m = (1 + \epsilon)t$ entries corresponding to each of the m rows. Each of the t lists are assigned to a random entry from $[m - w + 1]$. If the i-th list is assigned to entry $j \in [m - w + 1]$, then one of the w items in the list are placed into each of the entries $\{j, j + 1, \ldots, j + w - 1\}$. Note, the maximum load of any of m entries is equivalent to the largest consecutive section of non-zero entries in any of the m rows of the generated random band matrix after sorting by column starting location.

Prior work [Asharov et al. 2016] studied the setting where each of the t lists picked one of the m entries uniformly at random. We adapt the analysis for the slightly skewed distribution used for random band matrices in our work where only one of the first $m - w$ entries are chosen uniformly at random. Indeed, we prove the following theorem:

Theorem 4.4. *Consider a $m \times t$ matrix with $m = (1 + \epsilon)t$ where each column consists of a single random w-bit band. For constant $\epsilon > 0$ and band length $w = O(\lambda + \log t)$, the random band matrix has column rank n and executing Gaussian elimination after sorting the columns by the starting location of the band runs in time $O(tw)$ except with probability $2^{-\lambda}$.*

Proof of Theorem 4.4. We use the coupling described above. Therefore, it suffices for us to analyze only two-dimensional balls-into-bins allocations. We denote binary random variables $X_{i,j}$ to be whether the random band of the i-th column will overlap with the j-th row. Therefore, $X_{i,j} = 1$ if this event is true and $X_{i,j} = 0$ otherwise. Note, $X_{i,j} = 1$ if and only if the i-th column's random band starts in the set of row indices $\{j-w+1, j-w+2, \ldots, j\}$. In other words, $\mathsf{E}[X_{i,j} = 1] \leqslant w/(m-w+1)$. Let B_j be the total number of columns whose random bands overlap with the j-th row. By linearity of expectation, we get that

$$B_j = \sum_{i \in [t]} \mathsf{E}[X_{i,j}] \leqslant \frac{tw}{m - w + 1}.$$

Note that each $X_{i,j}$ is an independent random variable. Therefore, we can apply Chernoff bounds to get that

$$\Pr\left[B_j > 3 \cdot \frac{tw}{m-w+1}\right] \leqslant 2^{-tw/(m-w+1)}.$$

Next, we apply a Union bound over all m rows to get that B_j for all $j \in [m]$ is upper bounded by the same value with probability at most $m \cdot 2^{-tw/(m-w+1)}$. Finally, by noting that $m = (1+\epsilon)t$ for some constant $\epsilon > 0$ and picking $w = O(\lambda + \log t)$, we get that each row has a band length of at most $O(w)$ except with probability $2^{-\lambda}$. $\square$

We now formally present LSObvCompress using random band matrices. First we present $\pi_{\mathsf{LSObvCompress.ObvCompress}}$ and $\pi_{\mathsf{GenRandVec}}$.

Figure 4.4: Procedure $\pi_{\mathsf{LSObvCompress.ObvCompress}}$

Usage: On input additively homomorphic encrption public key $\mathbf{pk}_{\mathcal{E}}$, vector of n ciphertexts $\tilde{\mathbf{c}}$, number of non-zero plaintext entries t, and randomness R; output a compressed encoding, $\hat{\mathbf{c}}$, of $\tilde{\mathbf{c}}$.

```
1. $m \leftarrow (1+\epsilon)t$
2. $\mathbf{M} \leftarrow 0^{m \times n}$
3. For $i = 1, \ldots, n$
    (a) $\mathbf{v_i} \leftarrow \pi_{\mathsf{GenRandVec}}(i, m; R)$
    (b) $\mathbf{M}[:][i] \leftarrow \mathbf{v_i}$    (Set the $i$th column to $\mathbf{v_i}$)
4. $\hat{\mathbf{c}} \leftarrow \mathbf{M} \cdot \tilde{\mathbf{c}}$    (HE add using $\mathcal{E}$.Eval and $\mathbf{pk}_{\mathcal{E}}$)
5. Output $\hat{\mathbf{c}}$
```

Figure 4.5: Procedure $\pi_{\mathsf{GenRandVec}}$

Usage: On input column index i, column vector length m, and randomness R; output randomly generated column vector $\mathbf{v_i}$.

```
1. $w \leftarrow$ band width
2. $s \leftarrow \mathsf{H}_1(R \,||\, i)$    (Random value from $[m - w + 1]$)
3. $\mathbf{u} \leftarrow \mathsf{H}_1(R \,||\, i)$    (Random $w$-bit band.)
4. $\mathbf{v_i} \leftarrow 0^m$
5. For $j = 0, \ldots, w - 1$
    (a) $\mathbf{v_i}[s + j] \leftarrow \mathbf{u}[j]$
6. Output $\mathbf{v_i}$
```

Now we present $\pi_{\mathsf{LSObvCompress.Decompress}}$ and $\pi_{\mathsf{SolveLinearSystem}}$.

Figure 4.6: Procedure $\pi_{\mathsf{LSObvCompress.Decompress}}$

Usage: On input additively homomorphic encryption secret key $\mathbf{sk}_{\mathcal{E}}$, compressed encoding of ciphertexts $\hat{\mathbf{c}}$, set of non-zero plaintext indices I, and randomness R; output original non-zero plaintext vvalues $\{i, p_i\}_{i \in I}$.

```
1. $m \leftarrow (1 + \epsilon)t$
2. $\mathbf{M}_{\mathrm{c}(I)} \leftarrow 0^{m \times t}$    (Initialize all zero matrix.)
3. For $i_j \in I = \{i_1, \ldots, i_t\}$
    (a) $\mathbf{v}_{i_j} \leftarrow \pi_{\mathsf{GenRandVec}}(i_j, m; R)$
    (b) $\mathbf{M}_{\mathrm{c}(I)}[:][j] \leftarrow \mathbf{v}_{i_j}$    (Set the $j$th column to $\mathbf{v}_{i_j}$)
4. $\hat{\mathbf{p}} \leftarrow$ decryption of $\hat{\mathbf{c}}$ using $\mathcal{E}.\mathsf{Dec}$ and $\mathbf{sk}_{\mathcal{E}}$
5. $p_I \leftarrow \pi_{\mathsf{SolveLinearSystem}}(\mathbf{M}_{\mathrm{c}(I)}, \hat{\mathbf{p}})$
6. If $p_I = \perp$ then return $\perp$
```

```
7. $\mathbf{p} \leftarrow \emptyset$
8. For $i_j \in I = \{i_1, \ldots, i_t\}$: $\mathbf{p} \leftarrow \mathbf{p} \cup \{(i_j, (p_I)_j)\}$
9. Output $\mathbf{p}$
```

Figure 4.7: Procedure $\pi_{\mathsf{SolveLinearSystem}}$

Usage: On input: LHS matrix $\mathbf{M}$ and RHS values to solve for $\hat{\mathbf{p}}$; output solution to the linear system $\mathbf{M} \cdot \mathbf{p} = \hat{\mathbf{p}}$.

```
1. $(\mathbf{M}^\pi, \pi) \leftarrow$ column sorting of the matrix $\mathbf{M}$ in ascending band start positions, along with
   the corresponding permutation that produces the column sorted matrix (e.g. $\mathbf{M}^\pi[:][i] = \mathbf{M}[:][\pi(i)]$)
2. $\mathbf{p}^\pi \leftarrow$ execute Gaussian elimination on $\mathbf{M}^\pi$ and $\hat{\mathbf{p}}$, $\perp$ if no unique solution
3. If $\mathbf{p}^\pi = \perp$ then return $\perp$
4. $\mathbf{p} \leftarrow 0^t$
5. For $i = 1 \ldots t$: $\mathbf{p}[\pi(i)] \leftarrow p^\pi[i]$
6. Output $\mathbf{p}$
```

Next, we analyze the properties of LSObvCompress showing that it combines the good compression rates and efficient encoding/decoding times of our prior two attempts.

Failure Probability and Compression Rate. For failure probability, we note LSObvCompress fails only when $\mathbf{M}_{\mathsf{c}(I)}$ does not have a unique solution or that unique solution cannot be found. By Theorem 4.4, we know this occurs with probability at most $2^{-\lambda}$ assuming that $w = O(\lambda/\epsilon + \log n)$.

In our experiments, we will use concrete parameters for w and ϵ for various values of t to obtain 2^{-40} error probability. We point readers to Section 4.6.1 for more details. For the compression rate, our experiments show that ϵ may be as small as 0.05. As a result, LSObvCompress obtains compression rates that are only 5% larger than optimal.

Running Time. We start by analyzing the compression algorithm that computes the matrix multiplication of $\mathbf{M}$ and the input ciphertext vector $\tilde{\mathbf{c}} = (\tilde{c}_1, \ldots, \tilde{c}_n)^T$. As $\mathbf{M}$ is a binary matrix with at most nw non-zero entries, this can be performed using at most nw ciphertext-ciphertext additions. For decompression, we note that the main cost is solving the linear system $\mathbf{M}_{\mathsf{c}(I)}$ that requires $O(tw)$ time by Theorem 4.4 that is corroborated by our experiments (see Section 4.6.1).

Noise Growth for SHE. Recall that for the applications to batch PIR and labeled PSI, we will initialize LSObvCompress, using lattice-based SHE schemes. Therefore, noise growth is an important factor to consider. Suppose that the input ciphertexts $\tilde{\mathbf{c}} = (\tilde{c}_1, \ldots, \tilde{c}_n)^T$ each have error at most $\mathsf{Err}(\tilde{c}_i) \leqslant e$. We note that the compression algorithm requires computing the sum of at most w ciphertexts. Therefore, each ciphertext in the compressed output has error at most $O(w \cdot e)$ as ciphertext-ciphertext additions only incur linear noise growth (see Appendix 4.1.1 for more details). As decompression is done after decryption, we do not need to worry about noise growth for decompression.

4.2.4 Comparison with Sparse Random Linear Codes [Kaufman and Sudan 2007; Liu and Tromer 2022]

Liu and Tromer [Liu and Tromer 2022] implicitly study oblivious ciphertext compression. They observe that Sparse Random Linear Codes (SRLCs) [Kaufman and Sudan 2007], which use matrices $\mathbf{M} \in \mathbb{F}^{m \times n}$ where each column has a small number of non-zero entries drawn randomly from $\mathbb{F}$ can be used. However, each entry is sampled independently, in an unstructured way, which results in larger encodings than with LSObvCompress. Indeed, they show that such matrices can be sampled with full rank with high probability only if $m = O(t \log^2 t \log \lambda)$, which is larger than

$m = 1.05t$ of LSObvCompress. Moreover, because SRLCs are unstructured, Gaussian elimination takes $O(t^3)$ time, resulting in slower $O(t^3)$ decoding time, compared to the $O(t \cdot \lambda)$ decoding time of LSObvCompress. Finally, since SRLCs use elements drawn randomly from $\mathbb{F}$, when used with FHE, large parameters must be used to handle the noise when multiplying ciphertexts by these large elements. However, LSObvCompress only uses elements from $\{0, 1\}$, which means that ciphertexts are only added together, resulting in minimal noise growth.

4.3 Oblivious Ciphertext Decompression

We show that similar ideas that we used to solve the oblivious ciphertext compression problem may also be used to solve the oblivious ciphertext decompression problem. As a reminder, in this problem, the compressor is given a plaintext vector, $\mathbf{p} = (p_1, \ldots, p_n)^T$, and t relevant indices $I \subset [n]$. The goal is for the decompressor to decode ciphertexts $\tilde{\mathbf{c}} = (\tilde{c}_1, \ldots, \tilde{c}_n)^T$ such that $\tilde{c}_i$ is an encryption of p_i for all relevant indices $i \in I$. There are no requirements for any $i \notin I$.

Description of LSObvDecompress. Essentially, we will apply the ideas of LSObvCompress, but in reverse. That is, we will start with a matrix $\mathbf{M}$ of dimension $n \times m$ (that both the compressor and decompressor can generate based on shared randomness), where $m = (1 + \epsilon)t$ is the encoding length for some constant $\epsilon > 0$. Then, based on relevant row indices $I = \{i_1, \ldots, i_t\} \subset [n]$, the compressor will solve the linear system formed by a $(t \times m)$-dimensional sub-matrix $\mathbf{M}_{\mathsf{r}(I)}$ of $\mathbf{M}$ and vector $\mathbf{p}_I = (p_{i_1}, p_{i_2}, \ldots, p_{i_t})$, to obtain compressed plaintext vector $\hat{\mathbf{p}}$ of dimension m. Specifically, the compressor will solve the linear system for $\hat{\mathbf{p}}$ satisfying $\mathbf{M}_{\mathsf{r}(I)} \cdot \hat{\mathbf{p}} = \mathbf{p}_I$ using sub-matrix $\mathbf{M}_{\mathsf{r}(I)} = (\mathbf{M}_{i_1}^T, \ldots, \mathbf{M}_{i_t}^T)$.

Afterwards, the vector $\hat{\mathbf{p}}$ is encrypted entry-wise. The encrypted version of $\hat{\mathbf{p}}$ is the final encoding that we denote $\hat{\mathbf{c}}$. If the linear system according to $\mathbf{M}_{\mathsf{r}(I)}$ is not solvable, then the encoding fails and the compressor outputs any m encryptions. In applications, this is the point where we can utilize packing techniques where multiple plaintext values may be encrypted into a single ciphertext (as done in [Angel et al. 2018]).

For oblivious decompression, the decompressor computes $\mathbf{M} \cdot \hat{\mathbf{c}}$ homomorphically. Intuitively, this gives the decompressor ciphertext vector $\tilde{\mathbf{c}} = (\tilde{c}_1, \ldots, \tilde{c}_n)^T$ such that for each $i_j \in I$, the underlying plaintext of $\tilde{c}_{i_j}$ is $\mathbf{v}_{i_j} \cdot \hat{\mathbf{p}}$, which is exactly p_{i_j}, as desired. For every $\tilde{c}_{i_j}$ where $i_j \notin I$, the underlying plaintext will be some arbitrary linear combination of the entries of $\hat{\mathbf{p}}$, but recall that these values need not be correct.

For the choice of matrix $\mathbf{M}$, we can in fact generate it similarly as in LSObvCompress as a random band matrix of dimension $n \times m$, where each row consists of a single random band of length w. Note, this is the original random band matrix construction [Dietzfelbinger and Walzer 2019] without modification.

We now present the pseudocode for LSObvDecompress with procedures $\pi_{\text{LSObvDecompress.Compress}}$ and $\pi_{\text{LSObvDecompress.ObvDecompress}}$.

Figure 4.8: Procedure $\pi_{\text{LSObvDecompress.Compress}}$

Usage: On input additively homomorphic encryption secret key $\mathbf{sk}_{\mathcal{E}}$, plaintext values $\mathbf{p} = [p_1, \ldots, p_n]^T$, and randomness R; output: compressed ciphertexts $\hat{\mathbf{c}}$.

```
1. Compute $I = \{i \mid p_i \neq 0\} \subseteq [n]$.
2. If $|I| > t$, abort.
3. If $|I| < t$, arbitrarily add indices to $I$ until $|I| = t$.
4. $m \leftarrow (1 + \epsilon)t$
5. $\mathbf{M}_{\mathrm{r}(I)} \leftarrow 0^{t \times m}$    (Initialize all zero matrix.)
6. For $i \in I$: $\mathbf{M}_i \leftarrow \pi_{\text{GenRandVec}}(i, m, R)^T$
7. $\hat{\mathbf{p}} \leftarrow \pi_{\text{SolveLinearSystem}}(\mathbf{M}_{\mathrm{r}(I)}, \mathbf{p}_I)$
8. $\hat{\mathbf{c}} \leftarrow_\$$ encryption of $\hat{\mathbf{p}}$ using $\mathcal{E}$.Dec and $\mathbf{sk}_{\mathcal{E}}$
```

```
9. Output $\hat{\mathbf{c}}$
```

Figure 4.9: Procedure $\pi_{\text{LSObvDecompress.ObvDecompress}}$

Usage On input compressed ciphertexts $\hat{\mathbf{c}}$ and randomness R; output decompressed ciphertexts $\tilde{\mathbf{c}}$.

```
1. $m \leftarrow (1+\epsilon)t$
2. For $i \in [n]$: $\tilde{c}_i \leftarrow \pi_{\text{GenRandVec}}(i, m, R) \cdot \hat{\mathbf{c}}$
3. $\tilde{\mathbf{c}} \leftarrow (\tilde{c}_1, \ldots, \tilde{c}_n)$
4. Output $\tilde{\mathbf{c}}$
```

Failure Probability. From above, we saw that the encoding is correct as long as the compressor can solve the linear system associated with $\mathbf{M}_{\mathsf{r}(I)}$. For the failure probability, we can simply calculate the probability that $\mathbf{M}_{\mathsf{r}(I)}$ does have a unique solution (or it cannot be found). As $\mathbf{M}$ is a random band matrix, we know that $\mathbf{M}_{\mathsf{r}(I)}$ is also a random band matrix. Therefore, if we set the band length $w = O(\lambda/\epsilon + \log t)$ and $\mathbf{M}_{\mathsf{r}(I)}$ to be a $t \times (1+\epsilon)t$, then $\mathbf{M}_{\mathsf{r}(I)}$ has a unique solution.

Compression Rate. We show that ϵ may be as small as 0.05 using experimental evaluation (see Section 4.6.1). Note, this is only 5% larger than the minimum of t ciphertexts since t plaintext values must be correctly encoded.

Running Time. The compression algorithm requires solving the linear system associated to $\mathbf{M}_{\mathsf{r}(I)}$ that is a $t \times (1+\epsilon)t$ random band matrix. This can be done in $O(tw)$ time using only plaintext operations. Additionally, the resulting vector must be encrypted using $O(m) = O(t)$ time.

Decompression simply requires computing the matrix-vector multiplication $\mathbf{M} \cdot \hat{\mathbf{c}}$. As each row has at most w one entries, this requires $O(nw)$ homomorphic additions.

Obliviousness. Note that $\hat{\mathbf{c}}$ is always a length-m ciphertext vector. Reducing to the security of the underlying encryption scheme $\mathcal{E}$, we can replace each of these ciphertexts with encryptions of 0 meaning $\hat{\mathbf{c}}$ is independent of input plaintexts.

Noise Growth for SHE. Recall that for the applications to batch PIR and labeled PSI, we will initialize $\mathsf{LSObvCompress}$, using lattice-based SHE schemes. Therefore, noise growth is an important factor to consider. We note that the compression algorithm is performed in plaintext without any homomorphic operations. Therefore, we only consider noise growth for decompression. The compressed input consists of m fresh SHE ciphertexts. Decompression adds at most w ciphertexts. If the input ciphertexts $\tilde{\mathbf{c}} = (\tilde{c}_1, \ldots, \tilde{c}_m)^T$ have error $\mathsf{Err}(\tilde{c}_i) \leqslant e$ for all $i \in [m]$, then each output ciphertext has error at most $O(w \cdot e)$. See Appendix 4.1.1 for further details.

4.4 Batch PIR

We will present three single-server PIR schemes using our compression techniques. We refer readers to Section 4.6.2 for our experimental evaluation to choose the best option for various settings of database size, entry size and batch size. We also present an improved two-server scheme.

4.4.1 Single-Server: Compressed Responses

In this section, we present our improved single-server batch PIR with compressed responses that apply for large entries that cannot leverage vectorization techniques [Mughees and Ren 2023].

Cuckoo Hashing Batch PIR Framework. First, we review the Cuckoo Hashing Batch PIR Framework by Angel *et al.* [Angel et al. 2018]. In a naive batch PIR scheme, the server would process each of the ℓ queries on the entire n database entries, resulting in a total of $O(n\ell)$ server operations. To reduce server computation, Angel *et al.* [Angel et al. 2018] presented a batch PIR framework that cleverly utilizes cuckoo hashing to encode both the batch query and the database entries. To date, this is the most practically efficient approach to constructing a batch PIR scheme. Our batch PIR will be built directly from this framework.

In this framework, the server setup works by creating $B \geqslant \ell$ independent single-query PIR servers and replicating each of the n database entries appropriately to a subset of $\alpha \geqslant 1$ servers. Consider a sparse database $D = \{(k_1, v_1), \ldots, (k_n, v_n)\} \in (\mathcal{K} \times \mathcal{V})^n$. Concretely, the choice of the α-subset is determined by the individual database entry (k_i, v_i) and α independent hash functions $H_1, \ldots, H_\alpha : \mathcal{K} \to [B]$ mapping keys to one of the B servers. In particular, (k_i, v_i) will be replicated to the servers indexed by $H_1(k_i), \ldots, H_\alpha(k_i)$. The total number of entries across all B servers will be $n\alpha$.

The hash functions that will be shared between the client and the server so that the client may also perform batch queries. Given a batch query $\{q_1, \ldots, q_\ell\}$, the client performs cuckoo hashing to map the ℓ query keys into the B buckets. In particular, each bucket will contain at most one query key after cuckoo hashing. Then, the client constructs a single-query PIR request for each of the B PIR servers. For empty buckets, the client will construct dummy “zero" requests such that the response to a dummy request will be a ciphertext that encrypts zero. Concretely, $B - \ell$ dummy requests. Finally, the server will process the B independent single-query PIR requests and send B responses back to the client.

Concrete Instantiation. Angel *et al.* [Angel et al. 2018] empirically determined that setting $B = 1.5\ell$ and $\alpha = 3$ results in an appropriate balance between the failure probability of the client allocation procedure, and efficiency. We notice that the request and response size in the cuckoo hashing framework is larger than the naive approach. In the naive approach, the request and response size are merely ℓ ciphertexts whereas the framework requires 1.5ℓ ciphertexts. This is 50% larger than the number of responses in the naive approach.

Index PIR from SHE Composition. Now we recall the state-of-the-art for single-query index PIR (index means that the keys are just $1, \ldots, n$). Recent PIR schemes compose the two classes of SHE schemes to obtain fast computation and small communication. These include the more theoretical work of Gentry and Halevi [Gentry and Halevi 2019] as well as recent practical PIR schemes of OnionPIR [Mughees et al. 2021] and Spiral [Menon and Wu 2022]. These families of PIR schemes enable larger levels of recursion than prior works (such as [Angel et al. 2018; Ali et al. 2021]) due to their superior ablility to minimize noise growth by switching between SHE schemes.

At a high level, these PIR schemes operate as follows. The database is represented as a hypercube with dimension $d_1 \times d_2 \times \ldots \times d_z$. The first dimension is typically large such as $d_1 \in \{128, 512, 1024\}$ while the other dimensions are the same and much smaller $d_2 = \ldots = d_z \in \{2, 4\}$. For convenience, we will denote $d = d_2 = \ldots = d_z$. For a PIR request, the client will upload an encrypted vector of length $d_1 + d_2 + \ldots + d_z = d_1 + (z-1)d$ specifying a single entry in each of the z dimensions. The client uploads these as Regev/BFV encryptions using the packing techniques from Angel *et al.* [Angel et al. 2018] that can be unpacked by the server. Afterwards, the server applies the first dimension using Regev/BFV encryptions. The remaining Regev/BGV encryptions in the client's request are converted to GSW ciphertexts. Using the result of the first dimension's processing, the remaining $z - 1$ levels are handled using external products. Note, the final result is a Regev/BGV encryption that can be reduced using modulus switching before being returned the client.

Client Mapping or Keyword PIR. One subtlety of the cuckoo hashing batch PIR framework presented above is that each of the B independent single-query PIR servers consists of a sparse database. We note that this is true regardless of whether the original batch PIR problem consists of a dense database where $\mathcal{K} = [n]$ or a sparse database where $\mathcal{K}$ could be much larger. In earlier works (such as [Angel et al. 2018; Mughees and Ren 2023]), it was suggested to use $O(n)$ client mappings to convert from database indices to bucket indices. Recent work [Patel et al. 2023] instead directly uses state-of-the-art keyword PIR schemes to avoid linear client storage. At a high level, the only difference between [Patel et al. 2023] and the index PIR schemes presented above is that the client request may contain different inputs. However, the server-side processing, which is the only critical part needed when we analyze noise growth below, remains identical. Throughout the rest of this chapter, we will follow the approach of [Patel et al. 2023] and use single-query keyword PIR protocols for each of the B buckets.

Our Construction. To reduce communication in batch PIR, we will apply LSObvCompress to reduce the server response communication in the cuckoo hashing framework.

Namely, recall that for the $B - \ell$ buckets which do not have an associated key, the client will construct dummy "zero" requests such that the corresponding response ciphertext will encrypt zero. This can be done by setting, e.g., $\mathbf{v}_z$ to the zero-vector, since in the last step of the response algorithm, the server computes the inner product of $\mathbf{v}_z$ with some vector to obtain the final response ciphertext. Therefore, after the server processes the $B = 1.5\ell$ requests, it obtains $B = 1.5\ell$ responses of which ℓ consist of encrypted entries, and the rest are encrypted zeros. Thus, the server can apply the compression of LSObvCompress with $n = B$ and $t = \ell$ to obtain compressed ciphertexts. Of course, the client knows the indices of the ℓ real requests. As a result, the client can execute the decompression of LSObvCompress to obtain the requested entries.

Our construction therefore results in response size with overhead as small as $1.05\times$ the optimal, with minimal added computation, instead of the $1.5\times$ overhead in response size of [Angel et al. 2018].

Noise Growth of Prior (Keyword) PIR schemes. For noise growth, we first perform the noise analysis for prior PIR schemes using the keyword PIR framework [Patel et al. 2023] applied using recent PIR schemes from SHE composition [Mughees et al. 2021; Menon and Wu 2022] here. We will then see in the next section that our new PIR scheme increases the noise growth by a $O(w)$ multiplicative factor.

To compute the noise growth of this prior family of (keyword) PIR schemes, we will first make some assumptions without loss of generality. Suppose that the server has unpacked the request and obtained $d_1 + \ldots + d_z = d_1 + (z-1)d$ ciphertexts. We will assume that the d_1 Regev/BFV ciphertexts for the first level have error e_0. The remaining $(z-1)d$ GSW ciphertexts will have error e_1 that may be different due to translation to between Regev/BFV to GSW schemes. We will also assume that each database entry has norm at most ℓ. For large database entries, each entry is split into smaller parts each of norm at most ℓ.

Next, we can compute the noise growth of the above family of PIR schemes. We start by analyzing the first dimension processing where the result is n/d_1 BFV/Regev ciphertexts with error $O(d_1 \cdot \ell \cdot e_0)$ as each of the n/d_1 output ciphertexts are the result of summing $d_1 - 1$ ciphertext-plaintext multiplications with the original z BFV/Regev encryptions with error e_0.

We move onto the remaining $z-1$ dimensions. Consider the processing of the second dimension. The output will be $n/(d_1 \cdot d)$ ciphertexts where each ciphertext is the sum of d outputs from the external product operations. After the external product, the noise is $O(Be_1 + d_1\ell e_0)$. As we do d additions, the noise of each ciphertext after the second dimension processing is $O(dBe_1 + d_1 d\ell e_0)$. Repeating the analysis for all z dimensions, we obtain the noise of the final ciphertext is

$$O\left(\sum_{i=1}^{z-1} d^i Be_1 + \sum_{i=1}^{z-1} d^i d_1 \ell e_0\right).$$

Assuming that $d \geqslant 2$, we get that the final noise is

$$O((n/d_1)Be_1 + n\ell e_0)$$

since $d \geqslant 2$ and $d_1 \cdot d^{z-1} = \Theta(n)$.

Noise Analysis of Our PIR Scheme with Compression In this section, we analyze the noise growth of our PIR scheme that utilizes LSObvCompress for response compression. We will build on top of the analysis above. Recall that if we assume that the d_1 ciphertexts for the first dimension have error e_0 and the $(z-1)d$ ciphertexts for the other dimensions have error e_1, then the final ciphertext has noise $O((n/d_1)Be_1 + n\ell e_0)$.

Let us consider applying LSObvCompress additionally for response compression. Note, this would result in another $O(w)$ multiplicative factor in noise growth, since the algorithm adds together $O(w)$ ciphertexts at a time. Thus, the final noise is $O((n/d_1)Be_1w + n\ell e_0 w)$, which is only a $O(w)$ multiplicative factor larger than the prior schemes.

4.4.2 Single-Server: Compressed Requests

Next, we apply LSObvDecompress to compress requests for single-server batch PIR schemes.

Our Construction. In our framework using the keyword PIR from [Patel et al. 2023], the client generates $B = 1.5\ell$ requests, each containing z vectors. However, we only need correct answers from ℓ requests. Thus, the client can combine all $B \cdot z$ request vectors into one long vector, and for relevant indices I consisting only of entries corresponding to the z vectors for each of the ℓ important requests, apply LSObvDecompress. This will result in a compressed request with size overhead only $1.05\times$ compared to the naive batch PIR, which the client can then encrypt and send to the server. This is in contrast to the $1.5\times$ overhead in request size of [Angel et al. 2018]. We also utilize in our construction the request packing techniques from [Angel et al. 2018] to fit multiple requests into a single ciphertext.

The server will first apply the request ciphertext packing decoding and, then run decompression from LSObvDecompress to obtain the B encrypted requests. Note, only the ℓ important requests will be correct. This is sufficient as the remaining 0.5ℓ dummy requests are ignored by the client anyways. The remainder of the server processing and client decrypting remains identical.

Noise Growth. We show that applying LSObvDecompress increases noise by an $O(w)$ multiplicative factor. We will build on top of the analysis above. Since LSObvDecompress adds $O(w)$ ciphertexts together at a time, it is clear that the noise indeed increases by an $O(w)$ multiplicative factor to start. As a result, we can imagine that the ciphertexts now have error $O(we_0)$ and $O(we_1)$ respectively. After the server processes the z dimensions, the resulting ciphertext has noise $O((n/d_1)Be_1w + n\ell e_0 w)$. Therefore, if we only apply LSObvDecompress this increases the noise growth by a multiplicative $O(w)$ factor over the prior schemes.

4.4.3 Single-Server Batch PIR with Request and Response Compression

In this section, we present a single-server batch PIR scheme that uses both LSObvDecompress and LSObvCompress to compress requests and responses respectively. As we will see, this scheme is currently only of theoretical interest, as the noise growth is too large for practice.

We leverage one important aspect of the keyword PIR framework from [Patel et al. 2023], presented above. In this framework, the smallest dimension of the request vectors, say z w.l.o.g., is typically very small, of size $d_z = 2$ or $d_z = 4$. As described above, if the client wishes to construct a dummy "zero" request, it can set $\mathbf{v}_z$ to the all-0 vector of length 2 or 4, and all other vectors $\mathbf{v}_1, \ldots, \mathbf{v}_{z-1}$ arbitrarily. This is because when $\mathbf{v}_z$ is applied at the last level to obtain the final ciphertext, it will always produce a ciphertext that encrypts zero.

Our Construction. We use a similar approach as Section 4.4.1 and Section 4.4.2. However, we must modify request compression to be compatible with response compression. Recall that LSObvCompress for response compression requires that the dummy responses are zero encryptions.

To achieve this, we will set the last dimension of all 0.5ℓ dummy requests to be the all-zero vector. In the keyword PIR framework [Patel et al. 2023], the resulting response will be a zero encryption. The relevant indices become all ℓ real request vectors and the vector corresponding to the last dimension for each of the 0.5ℓ dummy requests. Afterwards, the client can execute LSObvDecompress to compress the request. The server decompresses the request using LSObvDecompress, processes the requests to compute $B = 1.5\ell$ responses and compresses using LSObvCompress as there are at most ℓ zero entries. Finally, the client decompresses to obtain the ℓ entries.

Efficiency. The client compresses a vector of length $1.5\ell \cdot (d_1 + \ldots + d_z)$ with $|I| = \ell \cdot (d_1 + \ldots + d_z) + 0.5\ell d_z$. Therefore, the compressed request consist of $\lceil 1.05|I|/r \rceil$ if r requests can fit into a single ciphertext using the request packing techniques in [Angel et al. 2018]. The compressed response has the identical size as the PIR scheme from Section 4.4.1.

Large Noise Growth Let us consider applying LSObvCompress additionally for response compression, after applying LSObvDecompress for request compression. It is easy to see taht this would result in another $O(w)$ multiplicative factor in noise growth, since the algorithm adds together $O(w)$ ciphertexts at a time. The final ciphertexts would therefore have noise $O((n/d_1)Be_1w^2 + n\ell e_0 w^2)$, which is $O(w^2)$ larger than before. This is in fact too large – we were unable to find parameters where request and response compression together beat either of the PIR schemes with just one of the techniques. We leave it as an open problem to find better SHE/PIR schemes enabling both request and response compression.

4.4.4 Single-Server: Vectorized Responses

We present a method to compress responses in conjunction with the recent vectorization techniques of Mughees and Ren [Mughees and Ren 2023]. The vectorization techniques [Mughees and Ren 2023] utilize *Single-Instruction-Multiple-Data* (SIMD) techniques. SIMD encodes multiple database entries into a single ciphertexts (leveraging additional structure of the SHE scheme) and operates on all of them simultaneously.

The core idea of utilizing LSObvCompress to compress responses remains the same, but we wish to leverage that multiple entries fit into a single ciphertext. To do this, we present a vectorized version of LSObvCompress that optimally packs multiple entries into a single ciphertext. If d entries fit into a single ciphertext, our vectorized LSObvCompress sends only $\lceil 1.05\ell/d \rceil$ to the client. In contrast, $\lceil 1.5\ell/d \rceil$ ciphertexts are encoded in [Mughees and Ren 2023].

At a high level, vectorized LSObvCompress works nearly identically as the LSObvCompress variant described in Section 4.2.3. The only difference is that we apply techniques from [Mughees and Ren 2023] to rotate ciphertexts and pack multiple entries into a ciphertext before performing compression. We start by presenting an overview of the prior vectorized batch PIR [Mughees and Ren 2023] before presenting our construction.

Vectorized Batch PIR. Mughees and Ren [Mughees and Ren 2023] proposed a batch PIR scheme that cleverly utilizes ciphertext vectorization to improve computation and reduce response sizes. In particular, they utilize *Single-Instruction-Multiple-Data* (SIMD) techniques for efficiently performing homomorphic operations [Smart and Vercauteren 2014]. SIMD encodes multiple database entries into a single ciphertext and operates on all of them simultaneously. In more detail, each plaintext polynomial consists of multiple SIMD slots in which multiple database entries can be encoded. Homomorphic operations can be applied to the ciphertexts and the plaintexts in SIMD fashion, i.e. single ciphertext-plaintext absorption corresponds to SIMD slot-wise ciphertext-plaintext absorption. Additionally, there are ciphertext rotation operations to manipulate and rotate the SIMD slots.

The vectorized batch PIR scheme [Mughees and Ren 2023] also builds on top of the cuckoo hashing based framework by Angel *et al.* [Angel et al. 2018]. In the cuckoo hashing framework, consider the point where the server completes processing the B PIR requests and holds the B PIR responses. Each valid response ciphertext will contain the desired entry in an arbitrary SIMD slot. The vectorized batch PIR merges multiple response ciphertexts using SIMD operations to reduce the total response size.

We now describe the merging of response ciphertexts in more detail. Let ℓ be the number of queries in the batch, $B = 1.5\ell$ be the number of single-query PIR buckets, and d be the number of SIMD slots. For simplicity, we will assume d to be a power of two and each database entry can be encoded in a single SIMD slot. Let $\tilde{c}_1, \ldots \tilde{c}_B$ be the response ciphertexts after the PIR servers process the queries, where each $\tilde{c}_i$ encrypts a length d vector of the form $(0, \ldots, 0, p_i, 0, \ldots, 0)$. Note that $\tilde{c}_i$ will contain at most one non-zero SIMD slot. If $\tilde{c}_i$ is the PIR response of a dummy request, then every slot of $\tilde{c}_i$ will be zero. Note that the server knows that at most ℓ PIR responses contain a non-zero slot and the remaining $B - \ell$ PIR responses will encrypt an entirely zero vector.

The vectorized scheme [Mughees and Ren 2023] converts $(0, \ldots, 0, p_i, 0, \ldots, 0)$ into $(p_i, \ldots, p_i)$ using ciphertext rotations. Without loss of generality, suppose that p_i is in the first slot. The server first rotates the ciphertext by 1 position to obtain a new ciphertext that contains p_i in the second slot. It then homomorphically adds the two ciphertexts to obtain a ciphertext that has p_i in the first two slots. Afterwards, it rotates the resulting ciphertext by 2 positions to obtain a new ciphertext that contains p_i in the third and the fourth slots. Again, it homomorphically adds the two ciphertexts to obtain a ciphertext that has p_i in the first four slots. Repeating this $\lceil \log_2 d \rceil$ times, we will obtain a ciphertext that has p_i in each of the d SIMD slots. This process works regardless of the original position of p_i. Afterwards, each $\tilde{c}_i = (p_i, \ldots, p_i)$ with the same value in every slot.

Next, the scheme masks each ciphertext to ensure that only a single, but predictable, slot may contain a non-zero value. The goal is to transform $\tilde{c}_i = (p_i, \ldots, p_i)$ to contain p_i only in the slot with i-th index. For example, we will multiply $\tilde{c}_1 \cdot (1, 0, \ldots, 0)$ to obtain $(p_1, 0, \ldots, 0)$. When $d < n$, we note that the i-th index would rotate from the last slot back to the first slot. For example, the $\tilde{c}_{d+1}$ will also be multiplied by $(1, 0, \ldots, 0)$ as the $(d + 1)$-th slot is equivalent to the first slot.

Afterwards, we get the following:

$$\begin{aligned}
\tilde{c}_1 &= (p_1, 0, 0, \ldots, 0) \\
\tilde{c}_2 &= (0, p_2, 0, \ldots, 0) \\
\ldots \\
\tilde{c}_d &= (0, 0, 0, \ldots, p_d) \\
\tilde{c}_{d+1} &= (p_{d+1}, 0, 0, \ldots, 0) \\
\tilde{c}_{d+2} &= (0, p_{d+2}, 0, \ldots, 0) \\
\ldots
\end{aligned}$$

Finally, merge each consecutive group of d ciphertexts into one by homomorphic additions to obtain the following:

$$(p_1, p_2, \ldots, p_d), (p_{d+1}, p_{d+2}, \ldots, p_{2d}), \ldots$$

Thus, this results in $\lceil B/d \rceil$ response ciphertexts.

The vectorized scheme by Mughees and Ren [Mughees and Ren 2023] can significantly reduce response size for small entries and large number of SIMD slots d. However, it still shares the same inefficiency as in Angel *et al.*'s framework. Out of the $B = 1.5\ell$ PIR requests, $B - \ell = 0.5\ell$ requests will correspond to dummy requests. Thus, 0.5ℓ PIR responses will be zero and at most ℓ will contain a non-zero slot. The vectorized scheme pessimistically encodes all 1.5ℓ responses requiring $\lceil B/d \rceil = \lceil 1.5\ell/d \rceil$ response ciphertexts. This means that these "dummy", zero responses must still occupy SIMD slots in the final responses. We show that we can apply LSObvCompress to effectively remove these dummy, zero PIR responses.

Our Construction. In the context of LSObvCompress, the input ciphertext vector consists of $n = B = 1.5\ell$ ciphertexts to compress, where there are at most $t = \ell$ ciphertexts that encrypt non-zero plaintext entries. Our goal is to compress this down to $(1+\epsilon)\ell$ SIMD slots, which implies we will need a total of $h = \lceil (1+\epsilon)\ell/d \rceil$ ciphertexts.

The main idea behind our usage of LSObvCompress stays unchanged from the large entry setting in Section 4.4.1. We present a vectorized version of LSObvCompress that efficiently packs multiple plaintext values into a single ciphertext. For each of the B response ciphertexts $\tilde{c}_i$, suppose we completed the first step of the vectorized PIR scheme [Mughees and Ren 2023]. Thus, all SIMD slots are populated with the plaintext entry, $\tilde{c}_i = (p_i, \ldots, p_i)$.

Next, we will apply our technique to compress using the matrix $\mathbf{M}$ that is described in Section 4.2.3 where $\mathbf{M}$ is the transpose of a random band matrix with dimensions $B \times m$ where $m = (1+\epsilon)\ell$. For convenience, we will denote the plaintext vector as $p = (p_1, \ldots, p_B)^T$. Using ideas from LSObvCompress, our goal is to compute the $m = (1+\epsilon)\ell$ SIMD slot values from the matrix-vector multiplication, $M \cdot p$. However, we would like to do this such that these are encoded in the slots of $h = \lceil m/d \rceil = \lceil (1+\epsilon)\ell/d \rceil$ ciphertexts.

At a high level, our vectorized version of LSObvCompress will encode the $m = (1+\epsilon)\ell$ SIMD slots such that the first d values of $M \cdot p$ will be in the first ciphertext, the next d values of $M \cdot p$ will be in the second ciphertext and so forth. We compute this vectorized encoding as follows. For each ciphertext $i \in [B]$, let $j_{i_1}, \ldots, j_{i_{g(i)}}$ be the indices of non-zero entries in the i-th column of $\mathbf{M}$. At a high level, we may imagine that the SIMD slots of the h ciphertexts are flattened, i.e. the left-hand-side column vector is of length $d \cdot h$. With this formulation, computing and adding $\mathbf{M}[:][i] \cdot p_i$ corresponds to adding the encryption of p_i to flattened slots with indices $j_{i_1}, \ldots, j_{i_{g(i)}}$. More precisely, for each $j_{i_k} \in \{j_{i_1}, \ldots, j_{i_{g(i)}}\}$, we compute a pair of indices $(a, b) = (\lfloor (j_{i_k} - 1)/d \rfloor + 1, (j_{i_k} - 1 \mod d) + 1)$, and homomorphically add to $\hat{\mathbf{c}}_a$ a multiplication of $\tilde{c}_i$ by a one-hot binary mask that is 1 only at the bth slot.

The decompression works identically, except the algorithm now flattens the decrypted SIMD slots to $(1+\epsilon)\ell$ individual plaintext entries before solving the linear system.

We formally present our vectorized version of LSObvCompress in $\pi_{\mathsf{LSObvCompress.VecObvCompress}}$. All differences with the original algorithm are highlighted in blue. We note the changes only enable vectorization and the core compression ideas remain identical. One can obtain the prior LSObvCompress algorithm by setting the number of SIMD slots to be $d = 1$.

Figure 4.10: Procedure $\pi_{\mathsf{LSObvCompress.ObvCompress}}$

Usage: On input: homomorphic encryption public key $\mathbf{pk}_{\mathcal{E}}$, vector of n SIMD ciphertexts with all slots populated with the same plaintext $\tilde{c}$, number of non-zero plaintexts t, and randomness R; output compressed ciphertexts $\hat{\mathbf{c}}$ of $\tilde{c}$.

```
1. $m \leftarrow (1+\epsilon)t$
2. $\mathbf{M} \leftarrow 0^{m\times n}$
3. For $i = 1, \dots, n$
    (a) $\mathbf{v_i} \leftarrow \pi_{\mathsf{GenRandVec}}(i, m; R)$    (Random column band)
    (b) $\mathbf{M}[:][i] \leftarrow \mathbf{v_i}$    (Set $i$-th column to $\mathbf{v_i}$)
4. $d \leftarrow$ number of SIMD slots per ciphertext
5. $\hat{\mathbf{c}} \leftarrow$ length $\lceil (1+\epsilon)t/d \rceil$ ciphertexts encrypting 0 slots
6. For $i = 1 \dots n$
    (a) For $j \in$ non-zero indices of $\mathbf{M}[:][i]$
        i. $a \leftarrow \lfloor (j-1)/d \rfloor + 1$
        ii. $b \leftarrow ((j-1) \mod d) + 1$
        iii. $msk \leftarrow$ one-hot binary mask with $b$-th slot set to 1
        iv. $\hat{\mathbf{c}}[a] \leftarrow \hat{\mathbf{c}}[a] + msk \cdot \tilde{c}[i]$
7. Output $\hat{\mathbf{c}}$
```

We point out that the compression algorithm will incur $O(B \cdot w)$ homomorphic operations (where w is the column band width), which is asymptotically the same running time as the original LSObvCompress with $B = n$ input ciphertexts. The decompression algorithm requires $O(\ell \cdot w)$ time.

4.4.5 Two-Server: Compressed Responses

Next, we use LSObvCompress to compress responses for two-server batch PIR. In this setting, the client sends requests to both servers. Each server holds a copy of the database and cannot communicate with each other. They then send individual responses back to the client, who uses both to reconstruct the requested entry. The same correctness and privacy conditions are required. Privacy is considered with respect to each individual server assuming non-collusion.

Two-Server PIR. To date, the most concretely efficient two-server PIR schemes are built using distributed point functions (DPF) [Gilboa and Ishai 2014; Boyle et al. 2016; Hafiz and Henry 2019]. A point function $f_i : \mathcal{K} \to \{0, 1\}$ satisfies $f_i(x) = 1$ if and only if $x = i$. DPFs enable secret sharing f_i amongst the two servers using two functions, f_i^0 and f_i^1, satisfying $f_i(x) = f_i^0(x) + f_i^1(x)$ for all $x \in \mathcal{K}$. Both f_i^0 and f_i^1 must not individually reveal anything about f_i.

To perform a two-server keyword PIR query for key k, the client uses DPFs to create secret shares of f_k, f_k^0 and f_k^1, that are sent to each of the two servers. Suppose the database consists of n key-value pairs, $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$. For each server $j \in \{0, 1\}$, the j-th server computes

$$z_j \leftarrow f_k^j(k_1) \cdot v_1 + \ldots + f_k^j(k_n) \cdot v_n.$$

Finally, the client receives z_0 and z_1 and computes the final answer $z_0 + z_1$. If $k = k_i$, then $z_0 + z_1 = v_i$. Otherwise, $z_0 + z_1 = 0$ when $k \notin \{k_1, \ldots, k_n\}$.

There is a small issue that the client cannot distinguish between $v_i = 0$ and $k \notin \{k_1, \ldots, k_n\}$.

To fix this, we can ensure that zero is not a valid entry. For example, one can simply append a 1-bit to the end of each entry, $v_1, \ldots, v_n$.

Two-Server Batch PIR. To our knowledge, the most efficient two-server batch PIR remains the cuckoo hashing framework of Angel *et al* [Angel et al. 2018]. In the concrete instantiation, we use a two-server, single-query, keyword PIR for each of the $B = 1.5\ell$ buckets when performing a batch query for ℓ entries.

Our Construction. Our goal is to utilize LSObvCompress to reduce the number of responses from $B = 1.5\ell$ that are sent by both servers. We will use the two-server keyword PIR based on the DPF of [Boyle et al. 2016]. Note that this PIR does not use encryption and, instead, relies on the non-collusion of the two servers for security. The encryption scheme $\mathcal{E}$ for LSObvCompress in this setting is additive secret sharing. Homomorphic additions will simply be addition operations by each server.

First, recall that in order to use LSObvCompress, the 0.5ℓ dummy requests must results in additive sharings (encryptions) of 0. We will add $(k_0, 0)$ is added to each of the B buckets for special key k_0. The client will issue a keyword PIR query for k_0 for each of the 0.5ℓ dummy buckets. The servers upon receipt of these requests for all B buckets will then compute the corresponding responses. Consider the i-th response z_i^0 and z_i^1 for both servers. Let $I' \subset [B]$ be the indices of the real, non-dummy requests. For all $i \in [B]$, $z_i = z_i^0 + z_i^1$ is the i-th requested entry. If $i \notin I'$, then $z_i = z_i^0 + z_i^1 = 0$.

We can thus apply LSObvCompress as follows. Both servers will use LSObvCompress to compress their responses $\mathbf{z}^0 = (z_1^0, \ldots, z_B^0)$ and $\mathbf{z}^1 = (z_1^1, \ldots, z_B^1)$. Recall that this is done by computing the matrix-vector multiplications $\hat{\mathbf{z}}^0 \leftarrow \mathbf{M} \cdot \mathbf{z}^0$ and $\hat{\mathbf{z}}^1 \leftarrow \mathbf{M} \cdot \mathbf{z}^1$ where $\mathbf{M}$ is the transpose of a random band matrix. The client will compute $\hat{\mathbf{z}} \leftarrow \hat{\mathbf{z}}^0 + \hat{\mathbf{z}}^1 = \mathbf{M} \cdot (\mathbf{z}^0 + \mathbf{z}^1)$ that is a compression of the requested entries, $\mathbf{z}^0 + \mathbf{z}^1$. Finally, the client runs the decompression portion of LSObvCompress on $\hat{\mathbf{z}}$ to obtain the non-dummy queried entries, z_i for all $i \in I$.

4.5 Labeled PSI

In this section, we show our techniques may be used to build protocols for labeled PSI in the unbalanced setting. Recall that the receiver has a set X and the sender has a labeled set $\{(y, L_y) \mid y \in Y\}$. Note, that one may interpret this as batch keyword PIR with the receiver as the client and the sender as the server. The only difference is that PSI requires privacy for both parties' inputs. Therefore, we need to also enable privacy for the sender (server) input. We present two improved constructions using our compression techniques.

4.5.1 Labeled PSI from Batch Keyword PIR and Oblivious PRF

We can use the generic composition from [Freedman et al. 2005] to build labeled PSI from batch keyword PIR and an oblivious pseudorandom function (OPRF). First, we describe the OPRF instantiation that we use for the transformation. We now present the formal oblivious PRF functionality used in our unbalanced labeled PSI protocol, $\mathcal{F}_{\text{oprf}}$.

Figure 4.11: Functionality $\mathcal{F}_{\text{oprf}}$

Parameters: There are two parties, a sender and a receiver. The receiver has set size n if honest and n' otherwise. Let $\text{out} \in \mathbb{Z}$ be the bit length.

Functionality:

1. The functionality first samples random function $F : \{0,1\}^* \to \{0,1\}^{\text{out}}$.
2. Subsequently, on input $(\textsc{Sender}, \text{sid}, Y)$ from the sender where $Y \subseteq \{0,1\}^*$, the functionality returns $\{F(y) \mid y \in Y\}$ to the sender.
3. Next, on input $(\textsc{Receive}, \text{sid}, X)$ from the receiver where $X \subseteq \{0,1\}^*$, ensure that $|X| \leq n$ if the receiver is honest and $|X| \leq n'$ otherwise. The functionality returns $(\textsc{Receiver-Input}, \text{sid})$ to the sender.
4. Thereafter, on input $(\textsc{Send}, \text{sid})$ from the sender, the functionality returns $\{F(x) \mid x \in X\}$ to

the receiver.

4.5.1.1 OPRF with Malicious Security

An OPRF allows the receiver to input set X and learn the set of pseudo-random outputs $\{F_k(x) \mid x \in X\}$, where F is a PRF, and k is known to the sender. For security, both the sender and receiver should learn nothing else (except the sender learns the size of X).

In this work, we will use the Diffie-Hellman based OPRF protocol of [Jarecki and Liu 2010] that computes the function $F_\alpha(x) = \mathsf{H}'(\mathsf{H}(x)^\alpha)$, where H, H' are hash functions modeled as random oracles. We take a description of this OPRF almost verbatim from [Chen et al. 2018]: Let G be a cyclic group with order q, where the One-More-Gap-Diffie-Hellman (OMGDH) problem is hard. H is a random oracle hash function with range $\mathbb{Z}_q^*$. The sender has a key $\alpha \in \mathbb{Z}_q^*$ and the receiver has a set of inputs X. In the OPRF request procedure $\pi_{\mathsf{OPRFRequest}}$, the receiver first samples $\beta_i \leftarrow \mathbb{Z}_q^*$ for each $i = 1 \ldots |X|$ and sends $\{\mathsf{H}(x_i)^{\beta_{x_i}} \mid i = 1 \ldots |X|\}$ to the sender.

Figure 4.12: Procedure $\pi_{\mathsf{OPRFRequest}}$

Usage: On input receiver's (ordered) set of items X; output list of temporary exponents $(\beta_1, \ldots, \beta_{|X|})$ and list of requests req.

```
1. req ← ⊥ For $i = 1 \ldots |X|$
    (a) $\beta_i \leftarrow_{\$} \mathbb{Z}_q^*$
    (b) req.insert($\mathsf{H}(X[i])^{\beta_i}$)
2. Output (($\beta_1, \ldots, \beta_{|X|}$), req)
```

Next, in the OPRF answer procedure $\pi_{\mathsf{OPRFAnswer}}$, the sender on input its PRF key α responds with $\{(\mathsf{H}(x_i)^{\beta_i})^\alpha \mid i = 1 \ldots |X|\}$.

Figure 4.13: Procedure $\pi_{\mathsf{OPRFAnswer}}$

Usage: On input PRF key α and list of requests req; output list of responses resp.

```
1. resp ← ⊥ For $i = 1 \ldots |\mathsf{req}|$: resp.insert($\mathsf{req}[i]^{\alpha}$)
2. Output resp
```

Finally, in the OPRF process procedure π_{OPRFProc}, the receiver outputs

$$\mathsf{H}'(\mathsf{H}(x_i)^{\alpha}) = \mathsf{H}'((\mathsf{H}(x_i)^{\beta_i})^{\alpha})^{1/\beta_i})$$

for each $x \in X$ (each output consists of a hash and an encryption key, to be used in the unbalanced labeled PSI protocol).

Figure 4.14: Procedure π_{OPRFProc}

Usage: On input receiver's (ordered) input set X, list of temporary exponents $(\beta_1, \ldots, \beta_{|X|})$ and list of responses resp; output mapping from items to OPRF outputs, X'.

```
1. $X' \leftarrow \perp$
2. For $i = 1 \ldots |\mathsf{resp}|$
    (a) $(\hat{x}_i, k_i) \leftarrow \mathsf{H}'(\mathsf{resp}[i]^{1/\beta_i})$
    (b) $X'.\mathsf{mapInsert}(X[i], (\hat{x}_i, k_i))$
3. Output $X'$
```

The outer hash function H' is used to map the group element to a sufficiently long bit string, and is modeled as a random oracle to help facilitate extraction in the malicious setting. In particular, by observing the queries made to $\mathsf{H}(x_i)$, the simulator can collect a list of pairs $\{(x_i, \mathsf{H}(x_i)\}$ which are known to the receiver. From this set the simulator can compute the set $A = \{(x_i, \mathsf{H}(x_i)^{\alpha})\}$. For some subset of the $\mathsf{H}(x_i)$, the receiver sends $\{\mathsf{H}(x_i)^{\beta_i}\}$ to the simulator, who sends back $\{\mathsf{H}(x_i)^{\beta_i\alpha}\}$.

For the receiver to learn the OPRF value for x_i, it must send $\mathsf{H}(x_i)^\alpha$ to the random oracle H'. At this time, the simulator extracts x_i if $(x_i, \mathsf{H}(x_i)^\alpha) \in A$. Although this OPRF does not facilitate extracting all x_i at the time the first message is sent, extraction is performed before the receiver learns the OPRF value, which will be sufficient for our purposes.

In the context of our unbalanced labeled PSI protocol, this OPRF has the property that the sender can use the same key with multiple receivers. This allows the sender, who has a large and often relatively static set, to pre-process its set only once.

4.5.1.2 Unbalanced Labeled PSI Transformation

Using a maliciously secure OPRF and a batch keyword PIR scheme, we now describe the generic construction of [Freedman et al. 2005] to get unbalanced labeled PSI. At a high-level, the sender and receiver use an OPRF to compute a (pseudo-random) hashed item and encryption key associated with each real item in their sets. The sender then builds a *pseudo-database* of key-value pairs with its hashed items as the keys and the encryption of the corresponding real labels under the corresponding encryption keys as the values. Then, the receiver uses the batch keyword PIR protocol to query for the hashed items in its set, and decrypts the labels using the corresponding encryption keys from the OPRF.

Procedure $\pi_{\mathsf{BuildPseudoDB}}$ formally describes the algorithm the sender uses to build their pseudo-database of hashed items and encrypted labels. First, the sender invokes $\mathcal{F}_{\mathsf{oprf}}$ on each $y \in Y$ and receive the output $(\hat{y}, k_y)$. Then from these outputs and the set of labels $\{L_y \mid y \in Y\}$ it constructs a pseudo-database $\mathsf{DB} = \{(\hat{y}, \mathsf{Enc}(k_y, L_y)) \mid y \in Y\}$.

Figure 4.15: Procedure $\pi_{\mathsf{BuildPseudoDB}}$

Usage: On input sender's set of items and associated labels $\{(y, L_y) \mid y \in Y\}$; output pseudo-database of hashed items and encrypted labels DB.

1. $\mathsf{DB} \leftarrow \perp$

```
2. Invoke $\mathcal{F}_{\text{oprf}}$ on input $Y$ to receive $\{(\hat{y}, k_y) \mid y \in Y\}$
3. For $(y, L_y) : y \in Y$
    (a) $\mathsf{ct}_y \leftarrow_\$ \mathsf{Enc}(k_y, L_y)$
    (b) $\mathsf{DB.mapInsert}(\hat{y}, \mathsf{ct}_y)$
4. Output DB
```

Next, the receiver's query procedure is formally described in $\pi_{\mathsf{ULPSIQuery}}$. The receiver invoke $\mathcal{F}_{\text{oprf}}$ on each input item x to receive $(\hat{x}, k_x)$ Then, it sends a batch keyword PIR request on input $Q' = \{\hat{x} \mid x \in X\}$.

Figure 4.16: Procedure $\pi_{\mathsf{ULPSIQuery}}$

Usage: On input receiver's input set of items X; output mapping from items to encryption keys K and batch keyword PIR request req.

```
1. $K \leftarrow \perp$
2. $Q \leftarrow \perp$
3. Invoke $\mathcal{F}_{\text{oprf}}$ on input $X$ to receive $\{(\hat{x}, k_x) \mid x \in X\}$
4. For $x \in X$
    (a) $K.\mathsf{mapInsert}(x, k_x)$
    (b) $Q.\mathsf{insert}(\hat{x})$
5. $\mathsf{req} \leftarrow_\$ \mathsf{BatchKWPIR.query}(Q)$
6. Output $(K, \mathsf{req})$
```

Then, the server's answer procedure, formally described in $\pi_{\mathsf{ULPSIAnswer}}$ answers the receiver's batch keyword PIR request using its pseudo-database DB.

Figure 4.17: Procedure $\pi_{\mathsf{ULPSIAnswer}}$

Usage: On input batch keyword PIR request req and pseudo-database DB); output batch keyword PIR response resp.

```
1. resp ← BatchKWPIR.answer(DB, req)
2. Output resp
```

Finally, the algorithm the receiver uses to decrypt the server's response is formally described in $\pi_{\mathsf{ULPSIDecrypt}}$. The receiver uses the batch keyword PIR to obtain the payload for each $\hat{x}$ that was in DB, $\mathsf{Enc}(k_x, L_x)$, and decrypts it with the corresponding encryption key k_x from the OPRF invocation to output $\{(x, L_x) \mid \hat{x} \in X' \cap Y'\} = \{(x, L_x) \mid x \in X \cap Y\}$.

Figure 4.18: Procedure $\pi_{\mathsf{ULPSIDecrypt}}$

Usage: On input mapping from items to encryption keys K and batch keyword PIR response resp; output: intersectoin items and associated labels int.

```
1. int ← ⊥
2. $\{(x, \mathsf{ct}_x) \mid x \in X \cap Y\} \leftarrow$ BatchKWPIR.decrypt(resp)
3. For $(x, \mathsf{ct}_x) : x \in X \cap Y$
    (a) $L_x \leftarrow \mathsf{Dec}(K[x], \mathsf{ct}_x)$
    (b) int.insert($(x, L_x)$)
4. Output int
```

Now, we prove the following theorem showing that our labeled PSI is private against a malicious sender and secure against a malicious receiver, as in prior works [Chen et al. 2018; Cong et al. 2021]. We note that there are subtleties in the security argument due to the usage of the keyword PIR construction [Patel et al. 2023]. In particular, the database encoding algorithm from that keyword PIR construction has non-negligible failure probability. One option is to ensure that the database encoding algorithm fails negligibly by increasing the band length parameter (using the analysis from [Bienstock et al. 2023c]). However, we show that this is unnecessary – since the input to the database encoding algorithm is just key-value pairs where each key is a random hash of the corresponding item in Y output by the OPRF and each value is a pseudo-random encryption of the label under a random key output by the OPRF, this failure probability is not a function of the sender's set Y and thus reveals nothing about it. Therefore, it suffices to use the keyword PIR construction unmodified with non-negligible encoding failures.

Theorem 4.5. ULPSI *securely realizes* $\mathcal{F}_{\text{ul-psi}}$ *with privacy against a malicious sender and security against a malicious receiver in the* $\mathcal{F}_{\text{oprf}}$*-hybrid model.*

Proof. We recall from [Freedman et al. 2005] that privacy against a malicious sender follows immediately since the adversary learns nothing from the invocation of $\mathcal{F}_{\text{oprf}}$, and what the adversary receives from the receiver from the keyword batch PIR invocation reveals nothing about its query (and thus its set X). Correctness in the real world if the sender is semi-honest easily follows from the correctness of the underlying keyword batch PIR as well as the correctness and security of $\mathcal{F}_{\text{oprf}}$: If $x \notin Y$, then $\hat{x}$ will not be a key of DB (except with negligible probability), due to the pseudo-randomness of the OPRF. Otherwise, if $x \in Y$ then $\hat{x}$ will be a key of DB from the correctness of the OPRF, and so the keyword batch PIR will return the corresponding encrytped label $\mathsf{Enc}(k_x, L_x)$, which the receiver can decrypt with k_x (also correctly computed from the OPRF) to get L_x.

We now prove security against a malicious receiver, by describing a simulator $\mathcal{S}$.

- $\mathcal{S}$ first emulates $\mathcal{F}_{\text{oprf}}$ and when the malicious receiver invokes it on input X, it returns random $\{(\hat{x}, k_x) \mid x \in X\}$.
- Then, it sends X to the unbalanced labeled PSI functionality, $\mathcal{F}_{\text{ul-psi}}$, and receives $\{(x, L_x) \mid x \in X \cap Y\}$.

- Finally, it builds a size n_Y pseudo-database DB using first the key-value pairs $\{(\hat{x}, \mathsf{Enc}(k_x, L_x)) \mid x \in X \cap Y\}$ and then $(r, \mathsf{Enc}(k, 0))$, for random r, k, for the remaining $n_Y - |X \cap Y|$. If it fails, it outputs $\perp$ to the receiver.

- Then, on input the keyword batch PIR request from the malicious receiver, the simulator returns the honest response using pseudo-database DB to the receiver.

To see why the simulation works, we proceed with a hybrid argument. Hybrid $\mathcal{H}_0$ is the real world. Hybrid $\mathcal{H}_1$ is the real world except for all $y \notin X$ (where this X is the set input by the receiver to $\mathcal{F}_{\text{oprf}}$), the receiver replaces the key-value pairs of y in DB with $(r, \mathsf{Enc}(k, 0))$ for random r, k. It is easy to see that $\mathcal{H}_0$ is indistinguishable from $\mathcal{H}_1$ because (i) r, k are outputs of $\mathcal{F}_{\text{oprf}}$ unknown to the receiver, and are thus uniformly random; (ii) by reducing to the security of the encryption scheme, replacing $\mathsf{Enc}(k, L_y)$ with $\mathsf{Enc}(k, 0)$ is indistinguishable. Note at this point that even though the keyword batch PIR database encoding algorithm may fail with non-negligible probability, we have shown that this failure is simply a function of the pseudorandom ciphertexts of the database, and not the underlying items $\{(y, L_y) \mid y \in Y \setminus X\}$. Observe that $\mathcal{H}_1$ is in fact the ideal (simulated) world, and thus the proof is complete.

□

4.5.2 Improving Oblivious Polynomial Evaluation with LSObvDecompress

Now we show that we can use LSObvDecompress to reduce the receiver-to-sender communication of the unbalanced labeled PSI scheme of [Cong et al. 2021]. We first provide an overview of their scheme.

4.5.2.1 Overview of [Cong et al. 2021]

As in the construction from batch keyword PIR and OPRF of Section 4.5.1, the receiver and sender first both run an OPRF on their items to obtain a hash and an encryption key, the latter of which the sender uses to encrypt the corresponding item label. The receiver thus obtains $X' = \{(\hat{x}, k_x) \mid x \in X\}$ and the sender obtains $\mathsf{DB} = \{(\hat{y}, \mathsf{Enc}(k_y, L_y)) \mid y \in Y\}$. Then, they use the same cuckoo hashing technique of Angel *et al.* [Angel et al. 2018] in the batch PIR setting, in which using three hash functions h_1, h_2, h_3, the sender places each item $\hat{y}$ of DB in three different bins out of $1.5 \cdot |X|$ total bins, and the receiver places each item $\hat{x}$ of X' in a single one of the $1.5 \cdot |X|$ total bins so that no bin has more than one item. Then, for every bin B in which there is at most one $\hat{x}$, the receiver and sender essentially compute the intersection $\hat{x} \cap \{\hat{y} \mid y \in Y \cap B\}$.

More specifically, the sender first interpolates the polynomial satisfying the following:

$$G(x) = \begin{cases} \mathsf{Enc}(k_y, L_y) & \text{if } x = \hat{y} : y \in Y \cap B \\ \text{random field element} & \text{otherwise} \end{cases}$$

Then, using a FHE scheme $\mathcal{E}$, the receiver encrypts $\hat{x}$ and sends the ciphertext to the sender (where the receiver encrypts 0 for empty bins), who returns the homomorphic evaluation of the polynomial G on $\hat{x}$. Next, the receiver first decrypts this returned FHE ciphertext, then using k_x from the OPRF output on x, attempts to decrypt the inner ciphertext. If AEAD is used for this ciphertext, then if $\hat{x} \in \{\hat{y} \mid y \in Y \cap B\}$, the receiver will obtain L_x; otherwise, the decryption will output $\perp$.

It is clear that correctness holds. For security, the sender only sees OPRF queries for $|X|$ inputs, which reveal nothing about X, and then $1.5 \cdot |X|$ FHE ciphertexts. The receiver only sees AEAD ciphertexts encrypted with unknown random keys (from the OPRF) for items that are not in its set.

4.5.2.2 Applying LSObvDecompress

As we observed with the cuckoo hashing framework for batch PIR, for the $0.5 \cdot |X|$ bins in which there is no $\hat{x}$, it does not matter what encrypted value the receiver gives to the sender. Thus, we can use LSObvDecompress to compress the $1.5 \cdot |X|$ total ciphertexts with respect to only the $|X|$ indices in which the corresponding bin has some $\hat{x}$, resulting in only $(1 + \varepsilon)$ encrypted values sent by the receiver. Since the sender ends up just getting $(1 + \epsilon) \cdot |X|$ ciphertexts, security holds as before. As we show in our experiments (see Section 4.6.1), ϵ can be as small as 0.05. Therefore, we may reduce the receiver-sender communication of the [Cong et al. 2021] scheme as much as 30% as long as the noise budget is sufficient. We point to Section 4.6.2 for our experimental evaluation.

Note that we do not apply LSObvCompress to the sender's responses. This is because it crucially relies on the fact that the plaintexts of irrelevant indices are 0. However, unless the sender interpolates and evaluates the polynomial

$$G'(x) = \begin{cases} \mathsf{Enc}(k_y, L_y) & \text{if } x = \hat{y} : y \in Y \cap B \\ 0 & \text{otherwise} \end{cases}$$

which would be of exponential degree and thus extremely inefficient to interpolate and evaluate, then the underlying plaintexts of irrelevant indices are unlikely to be 0.

4.6 Experimental Evaluation

We perform experimental evaluation for our new compression algorithms, LSObvDecompress and LSObvCompress, as well as their improvements to batch PIR and labeled PSI. Finally, we also benchmark our protocols for the real world application of anonymous messaging.

Experimental Setup. We implemented our experimental evaluations with around 3000 lines of C++ code. All our experiments are performed using Ubuntu PCs with 96 cores, 3.7 GHz Intel Xeon W-2135 and 128 GB of RAM with only single-threaded execution. The AVX2 and AVX-512 instruction sets with SIMD instructions are enabled. The results are the average of at least 10 experimental trials with standard deviation less than 10% of the averages. Our implementations will target error probability 2^{-40} and 128 bits of computational security. Server monetary costs are computed using Amazon EC2 savings plan pricing of t2.2xlarge instances [AWS 2023] of \$0.09 per GB of traffic and \$0.021 per CPU hour at the time. We will utilize SHA256 as the hash function and AES-GCM-256 as the encryption scheme with 32 byte keys. Unless otherwise specified, we will use the compression parameter $\epsilon = 0.05$ for our experiments.

Interpreting the Experimental Results. As our compression schemes are general schemes that can be instantiated on various protocols, they incur additional computational overhead compared to the ones that don't use our compression schemes. To assess concrete tradeoffs between the computational overhead and the communication reduction, we will use the Amazon EC2 server monetary cost model which measures the communication and computational efficiency as a dollar cost. We note that this model has been used in prior works [Patel et al. 2023; Bienstock et al. 2023c] for this exact purpose.

4.6.1 Oblivious Ciphertext Compression

We first evaluate the performance of LSObvCompress and LSObvDecompress in isolation and report results in Table 4.1. We also evaluate the performance of vectorized LSObvCompress in isolation and report the results in Table 4.2.

Setup. In our experiments for LSObvCompress and LSObvDecompress, we will use Regev encryption [Regev 2005] as the underlying scheme using the implementation from Spiral [Spiral 2022]. We fix the plaintext size to 8 KB and the ciphertext size to 20 KB.

We implement vectorized LSObvCompress using Microsoft SEAL [SEAL 2023] library. Following prior works [Mughees and Ren 2023], we use the polynomial degree of 8192, ciphertext modulus of 200 bits, and plaintext modulus of 20 bits for our experimental setup. We fix the entry size to 256 bytes.

Compression Size	Sizes & Schemes	Compression Time	Decompression Time	Total Time
1.05t	**t = 512, n = 768**			
	LSObvCompress	2.38 s	0.66 s	3.04 s
	LSObvDecompress	0.60 s	1.65 s	2.25 s
	t = 1024, n = 1536			
	LSObvCompress	5.15 s	1.34 s	6.49 s
	LSObvDecompress	1.25 s	3.94 s	5.19 s
	t = 2048, n = 3072			
	LSObvCompress	10.54 s	2.67 s	13.21 s
	LSObvDecompress	2.48 s	6.88 s	9.38 s
	t = 4096, n = 6144			
	LSObvCompress	21.45 s	5.27 s	26.72 s
	LSObvDecompress	5.05 s	14.50 s	19.55 s
1.07t	**t = 512, n = 768**			
	LSObvCompress	1.82 s	0.53 s	2.35 s
	LSObvDecompress	0.49 s	1.34 s	1.83 s
	t = 1024, n = 1536			
	LSObvCompress	4.12 s	1.07 s	5.19 s
	LSObvDecompress	0.96 s	3.19 s	4.15 s
	t = 2048, n = 3072			
	LSObvCompress	8.41 s	2.12 s	10.53 s
	LSObvDecompress	1.72 s	5.80 s	7.52 s
	t = 4096, n = 6144			
	LSObvCompress	17.06 s	4.43 s	21.49 s
	LSObvDecompress	2.76 s	11.19 s	13.95 s

Table 4.1: Evaluations of LSObvCompress and LSObvDecompress for different values of t (non-zero/relevant entries) and n (uncompressed input size). We fix the plaintext size to 8 KB and ciphertext size to 20 KB for all our results.

In the tables, t corresponds to the number of non-zero entries and n corresponds to the total number of entries including the zero entries. We fix the fraction of zero entries to $0.5t$ (thus $n = 1.5t$). Note that this corresponds to the fraction of dummy requests/responses in the cuckoo hashing framework from [Angel et al. 2018]. In our evaluations, we will target two compression sizes of $1.05t$ and $1.07t$. Note that this results in 30% and 29% request/response size reduction respectively.

Results. We see that computation time increases with better compression rate as well as larger t and n. However, we claim that our LSObvCompress and LSObvDecompress remain practically efficient for many applications; as we will show in the next sections, the additional computational cost is a relatively small fraction of the entire protocol's computation time for batch PIR, and the

Compression Size	Size	Compression Time	Decompression Time	Total Time
$1.05t$	**t = 512, n = 768**	12.8 s	0.1 s	12.9 s
	t = 1024, n = 1536	25.1 s	0.2 s	25.3 s
	t = 2048, n = 3072	50.5 s	0.5 s	60.0 s
	t = 4096, n = 6144	100.1 s	0.9 s	101.0 s
$1.07t$	**t = 512, n = 768**	10.3 s	0.1 s	10.4 s
	t = 1024, n = 1536	20.9 s	0.2 s	21.1 s
	t = 2048, n = 3072	41.6 s	0.4 s	42.0 s
	t = 4096, n = 6144	83.8 s	0.9 s	84.7 s

Table 4.2: Performance of vectorized LSObvCompress evaluated on various values of t and n. We use fixed entry size of 256 B.

significant reduction in the communication cost will justify these small additional computational overhead.

Note that the results in Table 4.2 correspond to ideal cases where the noise level budget is sufficient enough to accommodate additional homomorphic operations incurred by our scheme. In particular, this means that we may not be able to directly plug in our scheme to the vectorized PIR scheme [Mughees and Ren 2023], as different set of parameters may have to be chosen to handle the extra noise growth.

4.6.2 Single-Server Batch PIR

We evaluate the single-server batch PIR schemes from Section 4.4 using our compression techniques to reduce communication. We report our results in Table 4.3.

Setup. We implement our compression algorithms on top of the open-source Spiral implementation [Spiral 2022]. We use $n = 1$ million database entries for all of our results. Baseline corresponds to Angel *et al* [Angel et al. 2018]'s batch PIR framework implemented on top of Spiral [Menon and Wu 2022] without our compression techniques. The parameters for the baseline were chosen using the script provided by their open-source implementation. In our evaluations, we consider three batch sizes $\ell \in \{512, 1024, 2048\}$ (in the context of ciphertext compression/decompression,

DB Entry Size	Batch Size & Schemes	Public Param Size	Request Size	Response Size	Total Server Time	Amortized Server Time	Total Client Time	Server Monetary Cost
8 KB	ℓ = **512**							
	Baseline	20.87 MB	1.81 MB	15.59 MB	**840 s**	**1.64 s**	**6.1 s**	$0.00646
	LSObvCompress	22.62 MB	1.81 MB	**10.92 MB**	890 s	1.74 s	6.7 s	**$0.00633**
	LSObvDecompress	20.87 MB	**1.40 MB**	15.59 MB	863 s	1.69 s	6.4 s	$0.00656
	ℓ = **1024**							
	Baseline	20.87 MB	3.35 MB	31.18 MB	**1,256 s**	**1.23 s**	**6.8 s**	$0.01043
	LSObvCompress	23.11 MB	3.35 MB	**21.84 MB**	1,369 s	1.34 s	8.2 s	**$0.01025**
	LSObvDecompress	20.87 MB	**2.55 MB**	31.18 MB	1,323 s	1.29 s	8.0 s	$0.01075
	ℓ = **2048**							
	Baseline	20.87 MB	3.96 MB	62.37 MB	**1,750 s**	**0.85 s**	**7.0s**	$0.01617
	LSObvCompress	23.43 MB	3.96 MB	**43.67 MB**	1,871 s	0.91 s	9.7 s	**$0.01520**
	LSObvDecompress	20.87 MB	**3.15 MB**	62.37 MB	1,812 s	0.89 s	9.4 s	$0.01646
16 KB	ℓ = **512**							
	Baseline	20.87 MB	1.81 MB	31.18 MB	**1,286 s**	**2.51 s**	**6.3 s**	$0.01047
	LSObvCompress	22.62 MB	1.81 MB	**21.84 MB**	1,348 s	2.63 s	8.4 s	**$0.00999**
	LSObvDecompress	20.87 MB	**1.40 MB**	31.18 MB	1,308 s	2.55 s	8.3 s	$0.01056
	ℓ = **1024**							
	Baseline	20.87 MB	3.35 MB	62.37 MB	**1,775 s**	**1.73 s**	**7.9 s**	$0.01626
	LSObvCompress	23.11 MB	3.35 MB	**43.69 MB**	1,929 s	1.88 s	9.6 s	**$0.01548**
	LSObvDecompress	20.87 MB	**2.55 MB**	62.37 MB	1,881 s	1.83 s	9.4 s	$0.01681
	ℓ = **2048**							
	Baseline	20.87 MB	3.96 MB	124.74 MB	**2,634 s**	**1.29 s**	**8.5 s**	$0.02694
	LSObvCompress	23.43 MB	3.96 MB	**87.34 MB**	2,773 s	1.35 s	12.4 s	**$0.02439**
	LSObvDecompress	20.87 MB	**3.15 MB**	124.74 MB	2,746 s	1.34 s	12.4 s	$0.02752

Table 4.3: Evaluations of Spiral Batch PIR [Angel et al. 2018; Menon and Wu 2022] with and without our compression techniques, LSObvCompress and LSObvDecompress with $\epsilon = 0.05$. We fix the number of entries to $n = 1$ million for all our results.

the batch size ℓ corresponds to the number of non-zero entries). We follow the batch PIR setup from [Angel et al. 2018] and fix the fraction of dummy requests/responses (i.e. zero entries) to 0.5ℓ. We target compression size of 1.05ℓ.

Results. Using our LSObvCompress, we see 30% response size reduction in exchange for a reasonable additional computational cost compared to state-of-the-art PIR for large entries without compression. We see that this small additional computation cost is justified by the reduction in the server monetary cost. In particular, LSObvCompress reduces the server monetary cost by up to 10% compared to the baseline.

Using our LSObvDecompress algorithm, we see 20-24% reduction in the request size with slight increase in computation and server monetary cost.

We were unable to integrate our vectorized version of LSObvCompress into the vectorized batch PIR protocol [Mughees and Ren 2023] as we are unaware of an open-source implementation.

Batch Size & Schemes	**Response Size**	**Server Time**	**Client Time**	**Server Monetary Cost**
$\ell = 512$				
Baseline	221 KB	**9.63 s**	**0.01 s**	\$0.000076
LSObvCompress	**155 KB**	9.69 s	0.08 s	**\$0.000070**
$\ell = 1024$				
Baseline	442 KB	**9.76 s**	**0.01 s**	\$0.000096
LSObvCompress	**310 KB**	9.92 s	0.16 s	**\$0.000085**
$\ell = 2048$				
Baseline	885 KB	**9.79 s**	**0.01 s**	\$0.000136
LSObvCompress	**619 KB**	10.09 s	0.33 s	**\$0.000114**
$\ell = 4096$				
Baseline	1,769 KB	**9.81 s**	**0.01 s**	\$0.000216
LSObvCompress	**1,238 KB**	10.53 s	0.78 s	**\$0.000172**
$\ell = 8192$				
Baseline	3,539 KB	**9.82 s**	**0.01 s**	\$0.000375
LSObvCompress	**2,477 KB**	11.13 s	1.80 s	**\$0.000287**

Table 4.4: Comparison of DPF based two server batch PIR protocol [DPF 2021] with and without LSObvCompress ($\epsilon = 0.05$). We fix the number of database entries to $n = 1$ million and each entry size to 288 B for all our results.

Choosing the Right Protocol. In general, LSObvCompress provides the best communication and server monetary cost reduction. Thus, LSObvCompress will typically be the best option for most settings.

In certain settings, we note that LSObvDecompress may be useful where we wish to minimize upload communication from the client to the server. There are many natural settings where the upload costs/speed are more expensive/slower than the download costs/speed. For applications in these scenarios, it may be critical to save as much upload communication as possible that is achieved by LSObvDecompress.

4.6.3 Two-Server Batch PIR

We implement our response-compressed two-server batch PIR from Section 4.4.5 on top of the two-server single-query PIR implementation in [DPF 2021]. We report our results in Table 4.4.

Label Size & Schemes	Total Online Comm.	Total Online Time	Server Monetary Cost
512 B			
Cong *et al.* [Cong et al. 2021]	33.2 MB	**169 s**	\$0.00397
LSObvCompress	**11.4 MB**	304 s	**\$0.00279**
1024 B			
Cong *et al.* [Cong et al. 2021]	66.1 MB	**331 s**	\$0.00787
LSObvCompress	**11.6 MB**	355 s	**\$0.00311**
1536 B			
Cong *et al.* [Cong et al. 2021]	103.6 MB	535 s	\$0.01244
LSObvCompress	**11.9 MB**	**446 s**	**\$0.00367**

Table 4.5: Comparisons of Cong *et al.* [Cong et al. 2021]'s labeled PSI and our LSObvCompress-based PSI with $\epsilon = 0.05$. We fix the size of the sender's set to 1 million and the receiver's set to 512.

Setup. We fix the number of database entries to $n = 1$ million where each database entry is 288 bytes large. As in the single-server batch PIR experiment (Section 4.6.2), we fix the fraction of dummy responses to 0.5ℓ and target compression size of 1.05ℓ. We omit evaluating request sizes as they are the same for both schemes.

Results. We observe that using LSObvCompress can reduce response size by 30% in exchange for a small additional computational cost. However, the small additional computational cost is justified by the savings in the server monetary cost. Compared to the baseline, LSObvCompress can reduce the server monetary cost by up to 24%.

4.6.4 Labeled PSI

Next, we evaluate our labeled PSI built from our batch PIR in Section 4.4.1 and an OPRF protocol (see Appendix 4.5.1.1 for details). We report our results in Table 4.5.

Setup. Our implementation uses the single-server batch PIR implementation from Section 4.4.1 with the OPRF implementation from [APSI 2023]. We fix the size of the sender's set to 1 million and receiver's set to 512 (note, in the context of batch PIR these corresponds to the number of database elements and the batch size respectively). We have used one of the default parameter sets available in their open-source implementations for Cong *et al* [Cong et al. 2021]'s scheme.

Results. Our scheme has 65-88% reduced communication over prior state-of-the-art works [Cong et al. 2021]. For smaller label size, our construction with Spiral [Menon and Wu 2022] is slower, but we start to catch up and eventually outperform Cong *et al* [Cong et al. 2021] for larger label sizes. Note that even with these additional computation cost, we reduce the server monetary cost by 30-70%.

Due to the limitation of their open source implementation [APSI 2023], we could not compare our construction on larger label sizes, but we expect our scheme to outperform significantly as the label sizes increase. In any case, our communication cost and server monetary cost is significantly smaller.

5 | Conclusion

In this thesis, we make progress towards increasing the practical impact of efficient MPC protocols. First, we develop Fluid MPC protocols in the dishonest majority with preprocessing, honest majority, and two-thirds honest majority settings that have asymptotic communication complexity matching their counterparts in the traditional (non-Fluid) setting. Second, we develop batch PIR protocols with computational and communication complexity close to optimal. As a consequence, we get improved Private Join and Compute MPC protocols. We also get improved labeled PSI protocols.

Several open problems remain in both areas:

- For Fluid MPC, the question of efficient protocols with stronger output guarantees remains. In particular, efficient protocols that have security with identifiable abort (wherein at least one corrupt party is identified if an honest party aborts) are unknown for all three settings above. Moreover, for the honest and two-thirds honest majority settings, efficient protocols that have G.O.D. and even fairness are unknown.

- Also for Fluid MPC, the question of efficient protocols with *sub-optimal* corruption thresholds is unresolved. In the traditional setting, many protocols are known to have $O(|C|)$ communication overhead if the corruption threshold is $t < (1-\varepsilon)n/2$ for honest majority [Goyal et al. 2021b] and $t < (1-\varepsilon)n/3$ for two-thirds honest majority [Damgård et al. 2010]. It is currently unknown if such solutions exist for Fluid MPC.

- For batch PIR, as we mentioned earlier, using *both* oblivious ciphertext compression and oblivious ciphertext decompression is not advantageous in practice, as the increased communication costs associated with larger ciphertexts due to bigger noise from more homomorphic operations do not outweigh the decreased communication costs from the compression. Developing better somewhat-homomorphic encryption and/or single-query PIR schemes that can be used with both compression schemes so that overall communication is further decreased is an interesting open problem.

Bibliography

Abraham, I., Asharov, G., Patil, S., and Patra, A. (2023). Detect, pack and batch: Perfectly-secure MPC with linear communication and constant expected time. In Hazay, C. and Stam, M., editors, *EUROCRYPT 2023, Part II*, volume 14005 of *LNCS*, pages 251–281. Springer, Heidelberg.

Aguilar Melchor, C., Barrier, J., Fousse, L., and Killijian, M.-O. (2016). XPIR: Private information retrieval for everyone. *PoPETs*, 2016(2):155–174.

Ahmad, I., Yang, Y., Agrawal, D., El Abbadi, A., and Gupta, T. (2021). Addra: Metadata-private voice communication over fully untrusted infrastructure. In *OSDI 21*.

Ali, A., Lepoint, T., Patel, S., Raykova, M., Schoppmann, P., Seth, K., and Yeo, K. (2021). Communication-computation trade-offs in PIR. In Bailey, M. and Greenstadt, R., editors, *USENIX Security 2021*, pages 1811–1828. USENIX Association.

Angel, S., Chen, H., Laine, K., and Setty, S. T. V. (2018). PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society Press.

Angel, S. and Setty, S. (2016). Unobservable communication over fully untrusted infrastructure. In *OSDI 16*, pages 551–569.

APSI (2023). APSI: C++ library for Asymmetric PSI. `https://github.com/microsoft/APSI`.

Asharov, G., Naor, M., Segev, G., and Shahaf, I. (2016). Searchable symmetric encryption: optimal locality in linear space via two-dimensional balanced allocations. In Wichs, D. and Mansour, Y., editors, *48th ACM STOC*, pages 1101–1114. ACM Press.

AWS (2023). EC2 On-Demand Pricing. `https://aws.amazon.com/ec2/pricing/on-demand/`.

Badrinarayanan, S., Jain, A., Manohar, N., and Sahai, A. (2020). Secure mpc: laziness leads to god. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 120–150. Springer.

Beaver, D. (1992). Efficient multiparty protocols using circuit randomization. In Feigenbaum, J., editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, Heidelberg. Springer Berlin Heidelberg.

Beerliová-Trubíniová, Z. and Hirt, M. (2008). Perfectly-secure mpc with linear communication complexity. In Canetti, R., editor, *Theory of Cryptography*, pages 213–230, Berlin, Heidelberg. Springer Berlin Heidelberg.

Beimel, A., Ishai, Y., and Malkin, T. (2000). Reducing the servers computation in private information retrieval: PIR with preprocessing. In Bellare, M., editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 55–73. Springer, Heidelberg.

Beimel, A., Ishai, Y., and Malkin, T. (2004). Reducing the servers' computation in private information retrieval: PIR with preprocessing. *Journal of Cryptology*, 17(2):125–151.

Ben-Efraim, A., Nielsen, M., and Omri, E. (2019). Turbospeedz: Double your online spdz! improving spdz using function dependent preprocessing. In *International Conference on Applied Cryptography and Network Security*, pages 530–549. Springer.

Ben-Or, M., Goldwasser, S., and Wigderson, A. (1988). Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press.

Bendlin, R., Damgård, I., Orlandi, C., and Zakarias, S. (2011). Semi-homomorphic encryption and multiparty computation. In Paterson, K. G., editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 169–188. Springer.

Bienstock, A., Escudero, D., and Polychroniadou, A. (2023a). On linear communication complexity for (maximally) fluid mpc. In Handschuh, H. and Lysyanskaya, A., editors, *Advances in Cryptology – CRYPTO 2023*, pages 263–294, Cham. Springer Nature Switzerland.

Bienstock, A., Escudero, D., and Polychroniadou, A. (2023b). Perfectly secure fluid mpc with abort and linear communication complexity.

Bienstock, A., Patel, S., Seo, J. Y., and Yeo, K. (2023c). Near-optimal oblivious key-value stores for efficient PSI, PSU and volume-hiding multi-maps. In *USENIX Security 2023*.

Bienstock, A., Patel, S., Seo, J. Y., and Yeo, K. (2024). Batch pir and labeled psi with oblivious ciphertext compression. In *USENIX Security 2024*.

Boyle, E., Gilboa, N., and Ishai, Y. (2016). Function secret sharing: Improvements and extensions. In Weippl, E. R., Katzenbeisser, S., Kruegel, C., Myers, A. C., and Halevi, S., editors, *ACM CCS 2016*, pages 1292–1303. ACM Press.

Boyle, E., Gilboa, N., Ishai, Y., and Nof, A. (2020). Efficient fully secure computation via distributed zero-knowledge proofs. In *Advances in Cryptology – ASIACRYPT 2020*, pages 244–276, Cham. Springer International Publishing.

Brakerski, Z. (2012). Fully homomorphic encryption without modulus switching from classical GapSVP. In Safavi-Naini, R. and Canetti, R., editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Heidelberg.

Bretagnolle, J. and Huber, C. (1978). Estimation des densités : Risque minimax. In Dellacherie, C., Meyer, P. A., and Weil, M., editors, *Séminaire de Probabilités XII*, pages 342–363, Berlin, Heidelberg. Springer Berlin Heidelberg.

Cachin, C., Micali, S., and Stadler, M. (1999). Computationally private information retrieval with polylogarithmic communication. In Stern, J., editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 402–414. Springer, Heidelberg.

Canetti, R. (2001). Universally composable security: A new paradigm for cryptographic protocols. In *FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society.

Chaum, D., Crépeau, C., and Damgård, I. (1987). Multiparty unconditionally secure protocols (abstract). In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*, page 462.

Chen, H., Huang, Z., Laine, K., and Rindal, P. (2018). Labeled PSI from fully homomorphic encryption with malicious security. In Lie, D., Mannan, M., Backes, M., and Wang, X., editors, *ACM CCS 2018*, pages 1223–1237. ACM Press.

Chen, H., Laine, K., and Rindal, P. (2017). Fast private set intersection from homomorphic encryption. In Thuraisingham, B. M., Evans, D., Malkin, T., and Xu, D., editors, *ACM CCS 2017*, pages 1243–1255. ACM Press.

Chida, K., Genkin, D., Hamada, K., Ikarashi, D., Kikuchi, R., Lindell, Y., and Nof, A. (2018). Fast large-scale honest-majority mpc for malicious adversaries. In *Annual International Cryptology Conference*, pages 34–64. Springer.

Chillotti, I., Gama, N., Georgieva, M., and Izabachène, M. (2020). Tfhe: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91.

Choi, S. G., Dachman-Soled, D., Gordon, S. D., Liu, L., and Yerukhimovich, A. (2021). Compressed oblivious encoding for homomorphically encrypted search. In Vigna, G. and Shi, E., editors, *ACM CCS 2021*, pages 2277–2291. ACM Press.

Chor, B., Kushilevitz, E., Goldreich, O., and Sudan, M. (1998). Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981.

Choudhuri, A. R., Goel, A., Green, M., Jain, A., and Kaptchuk, G. (2021). Fluid MPC: Secure multiparty computation with dynamic participants. In Malkin, T. and Peikert, C., editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 94–123, Virtual Event. Springer, Heidelberg.

Cleve, R. (1986). Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th ACM STOC*, pages 364–369. ACM Press.

Cong, K., Moreno, R. C., da Gama, M. B., Dai, W., Iliashenko, I., Laine, K., and Rosenberg, M. (2021). Labeled PSI from homomorphic encryption with reduced computation and communication. In Vigna, G. and Shi, E., editors, *ACM CCS 2021*, pages 1135–1150. ACM Press.

Contact Discovery (2017). Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/.

Damgård, I., Escudero, D., and Polychroniadou, A. (2021). Phoenix: Secure computation in an unstable network with dropouts and comebacks. *Cryptology ePrint Archive*.

Damgård, I., Ishai, Y., and Krøigaard, M. (2010). Perfectly secure multiparty computation and the computational overhead of cryptography. In Gilbert, H., editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 445–465. Springer, Heidelberg.

Damgård, I. and Jurik, M. (2001). A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In Kim, K., editor, *PKC 2001*, volume 1992 of *LNCS*, pages 119–136. Springer, Heidelberg.

Damgård, I. and Nielsen, J. B. (2007). Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference*, pages 572–590. Springer.

Damgård, I., Pastro, V., Smart, N. P., and Zakarias, S. (2012). Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini, R. and Canetti, R., editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer.

David, B., Deligios, G., Goel, A., Ishai, Y., Konring, A., Kushilevitz, E., Liu-Zhang, C.-D., and Narayanan, V. (2023). Perfect MPC over layered graphs. In Handschuh, H. and Lysyanskaya, A., editors, *CRYPTO 2023, Part I*, volume 14081 of *LNCS*, pages 360–392. Springer, Heidelberg.

Demmler, D., Rindal, P., Rosulek, M., and Trieu, N. (2018). Pir-psi: Scaling private contact discovery. *Proceedings on Privacy Enhancing Technologies*, 2018(4):159–178.

Dietzfelbinger, M. and Walzer, S. (2019). Efficient gauss elimination for near-quadratic matrices with one short random block per row, with applications. In *27th Annual European Symposium on Algorithms (ESA 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

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

DPF (2021). C++ DPF-PIR library. https://github.com/dkales/dpf-cpp.

Dvir, Z. and Gopi, S. (2015). 2-server PIR with sub-polynomial communication. In Servedio, R. A. and Rubinfeld, R., editors, *47th ACM STOC*, pages 577–584. ACM Press.

Efremenko, K. (2009). 3-query locally decodable codes of subexponential length. In Mitzenmacher, M., editor, *41st ACM STOC*, pages 39–44. ACM Press.

Escudero, D., Goyal, V., Polychroniadou, A., and Song, Y. (2022). Turbopack: Honest majority MPC with constant online communication. ACM Conference on Computer and Communications Security (CCS).

Fan, J. and Vercauteren, F. (2012). Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144. https://eprint.iacr.org/2012/144.

Fitzi, M., Hirt, M., and Maurer, U. (1998). Trading correctness for privacy in unconditional multi-party computation. In *Annual International Cryptology Conference*, pages 121–136. Springer.

Fleischhacker, N., Larsen, K. G., and Simkin, M. (2023). How to compress encrypted data. In *EUROCRYPT 2023, Part I*, pages 551–577. Springer.

Franklin, M. K. and Yung, M. (1992). Communication complexity of secure computation (extended abstract). In *24th ACM STOC*, pages 699–710. ACM Press.

Freedman, M. J., Ishai, Y., Pinkas, B., and Reingold, O. (2005). Keyword search and oblivious pseudorandom functions. In Kilian, J., editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg.

Garimella, G., Pinkas, B., Rosulek, M., Trieu, N., and Yanai, A. (2021). Oblivious key-value stores and amplification for private set intersection. In Malkin, T. and Peikert, C., editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 395–425, Virtual Event. Springer, Heidelberg.

Genkin, D., Ishai, Y., Prabhakaran, M. M., Sahai, A., and Tromer, E. (2014). Circuits resilient to additive attacks with applications to secure computation. In *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*, STOC '14, pages 495–504, New York, NY, USA. ACM.

Gentry, C. and Halevi, S. (2019). Compressible FHE with applications to PIR. In Hofheinz, D. and Rosen, A., editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 438–464. Springer, Heidelberg.

Gentry, C., Halevi, S., Krawczyk, H., Magri, B., Nielsen, J. B., Rabin, T., and Yakoubov, S. (2021a). Yoso: you only speak once. In *Annual International Cryptology Conference*, pages 64–93. Springer.

Gentry, C., Halevi, S., Krawczyk, H., Magri, B., Nielsen, J. B., Rabin, T., and Yakoubov, S. (2021b). YOSO: you only speak once - secure MPC with stateless ephemeral roles. In *CRYPTO 2021*.

Gentry, C. and Ramzan, Z. (2005). Single-database private information retrieval with constant communication rate. In Caires, L., Italiano, G. F., Monteiro, L., Palamidessi, C., and Yung, M., editors, *ICALP 2005*, volume 3580 of *LNCS*, pages 803–815. Springer, Heidelberg.

Gentry, C., Sahai, A., and Waters, B. (2013). Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Canetti, R. and Garay, J. A., editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 75–92. Springer, Heidelberg.

Gertner, Y., Ishai, Y., Kushilevitz, E., and Malkin, T. (1998). Protecting data privacy in private information retrieval schemes. In *30th ACM STOC*, pages 151–160. ACM Press.

Gilboa, N. and Ishai, Y. (2014). Distributed point functions and their applications. In Nguyen, P. Q. and Oswald, E., editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, Heidelberg.

Goldreich, O., Micali, S., and Wigderson, A. (1987). How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229.

Goyal, V., Li, H., Ostrovsky, R., Polychroniadou, A., and Song, Y. (2021a). Atlas: efficient and scalable mpc in the honest majority setting. In *Annual International Cryptology Conference*, pages 244–274. Springer.

Goyal, V., Liu, Y., and Song, Y. (2019). Communication-efficient unconditional mpc with guaranteed output delivery. In *Annual International Cryptology Conference*, pages 85–114. Springer.

Goyal, V., Polychroniadou, A., and Song, Y. (2021b). Unconditional communication-efficient MPC via hall's marriage theorem. In Malkin, T. and Peikert, C., editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 275–304, Virtual Event. Springer, Heidelberg.

Goyal, V. and Song, Y. (2020). Malicious security comes free in honest-majority mpc. Cryptology ePrint Archive, Report 2020/134. https://eprint.iacr.org/2020/134.

Groth, J., Kiayias, A., and Lipmaa, H. (2010). Multi-query computationally-private information retrieval with constant communication rate. In Nguyen, P. Q. and Pointcheval, D., editors, *PKC 2010*, volume 6056 of *LNCS*, pages 107–123. Springer, Heidelberg.

Guo, Y., Pass, R., and Shi, E. (2019). Synchronous, with a chance of partition tolerance. In Boldyreva, A. and Micciancio, D., editors, *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part I*, volume 11692 of *Lecture Notes in Computer Science*, pages 499–529. Springer.

Hafiz, S. M. and Henry, R. (2019). A bit more than a bit is more than a bit better: Faster (essentially) optimal-rate many-server PIR. *PoPETs*, 2019(4):112–131.

Henry, R. (2016). Polynomial batch codes for efficient IT-PIR. *PoPETs*, 2016(4):202–218.

Ion, M., Kreuter, B., Nergiz, A. E., Patel, S., Saxena, S., Seth, K., Raykova, M., Shanahan, D., and Yung, M. (2020). On deploying secure computing: Private intersection-sum-with-cardinality. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 370–389.

Ishai, Y., Kushilevitz, E., Ostrovsky, R., and Sahai, A. (2004). Batch codes and their applications. In Babai, L., editor, *36th ACM STOC*, pages 262–271. ACM Press.

Jarecki, S. and Liu, X. (2010). Fast secure computation of set intersection. In Garay, J. A. and Prisco, R. D., editors, *SCN 10*, volume 6280 of *LNCS*, pages 418–435. Springer, Heidelberg.

Kales, D., Rechberger, C., Schneider, T., Senker, M., and Weinert, C. (2019). Mobile private contact discovery at scale. In Heninger, N. and Traynor, P., editors, *USENIX Security 2019*, pages 1447–1464. USENIX Association.

Kaufman, T. and Sudan, M. (2007). Sparse random linear codes are locally decodable and testable. In *48th FOCS*, pages 590–600. IEEE Computer Society Press.

Kushilevitz, E. and Ostrovsky, R. (1997). Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th FOCS*, pages 364–373. IEEE Computer Society Press.

Lamport, L., Shostak, R., and Pease, M. (1982). The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401.

Lepoint, T., Patel, S., Raykova, M., Seth, K., and Trieu, N. (2021). Private join and compute from PIR with default. In Tibouchi, M. and Wang, H., editors, *ASIACRYPT 2021, Part II*, volume 13091 of *LNCS*, pages 605–634. Springer, Heidelberg.

Lipmaa, H. (2005). An oblivious transfer protocol with log-squared communication. In *International Conference on Information Security*, pages 314–328. Springer.

Liu, Z. and Tromer, E. (2022). Oblivious message retrieval. In Dodis, Y. and Shrimpton, T., editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 753–783. Springer, Heidelberg.

Lueks, W. and Goldberg, I. (2015). Sublinear scaling for multi-client private information retrieval. In Böhme, R. and Okamoto, T., editors, *FC 2015*, volume 8975 of *LNCS*, pages 168–186. Springer, Heidelberg.

Mahdavi, R. A. and Kerschbaum, F. (2022). Constant-weight PIR: Single-round keyword PIR via constant-weight equality operators. In *USENIX Security 22*, pages 1723–1740, Boston, MA.

Menon, S. J. and Wu, D. J. (2022). Spiral: Fast, high-rate single-server PIR via FHE composition. In *2022 IEEE Symposium on Security and Privacy.*

MPC Deployments (2023). Mpc deployments. https://mpc.cs.berkeley.edu/.

Mughees, M. and Ren, L. (2023). Vectorized batch private information retrieval. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 437–452.

Mughees, M. H., Chen, H., and Ren, L. (2021). OnionPIR: Response efficient single-server PIR. In Vigna, G. and Shi, E., editors, *ACM CCS 2021*, pages 2292–2306. ACM Press.

Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In Stern, J., editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Heidelberg.

Park, J. and Tibouchi, M. (2020). SHECS-PIR: Somewhat homomorphic encryption-based compact and scalable private information retrieval. In Chen, L., Li, N., Liang, K., and Schneider, S. A., editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*, pages 86–106. Springer, Heidelberg.

Password Checkup (2019). Protect your accounts from data breaches with password checkup. https://security.googleblog.com/2019/02/protect-your-accounts-from-data.html.

Password Monitor (2021). Password monitor: Safeguarding passwords in microsoft edge. https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/.

Patel, S., Seo, J. Y., and Yeo, K. (2023). Don't be dense: Efficient keyword PIR for sparse databases. In *USENIX Security 2023*.

Polychroniadou, A., Asharov, G., Diamond, B., Balch, T., Buehler, H., Hua, R., Gu, S., Gimler, G., and Veloso, M. (2023). Prime match: A Privacy-Preserving inventory matching system. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6417–6434, Anaheim, CA. USENIX Association.

Private Join and Compute (2019). Helping organizations do more without collecting more data. https://security.googleblog.com/2019/06/helping-organizations-do-more-without-collecting-more-data.html.

Rabin, T. and Ben-Or, M. (1989). Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *21st ACM STOC*, pages 73–85. ACM Press.

Rachuri, R. and Scholl, P. (2022). Le mans: Dynamic and fluid MPC for dishonest majority. In Dodis, Y. and Shrimpton, T., editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 719–749. Springer, Heidelberg.

Regev, O. (2005). On lattices, learning with errors, random linear codes, and cryptography. In Gabow, H. N. and Fagin, R., editors, *37th ACM STOC*, pages 84–93. ACM Press.

SEAL (2023). Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA.

Smart, N. P. and Vercauteren, F. (2014). Fully homomorphic simd operations. *Designs, codes and cryptography*, 71:57–81.

Spiral (2022). Spiral. https://github.com/menonsamir/spiral.

Wallet as a Service (2023). Wallets as a service. https://www.coinbase.com/cloud/products/waas.

Yao, A. C. (1986). How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167.

Yeo, K. (2023). Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. In Handschuh, H. and Lysyanskaya, A., editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 197–230. Springer, Heidelberg.