Practical Multiparty Computation:

Approaches to Private Machine Learning

Rahul Rachuri

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Practical Multiparty Computation:
Approaches to Private Machine Learning

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Rahul Rachuri
November 30, 2022

Abstract

Machine Learning is becoming increasingly relevant, with widespread impact across industries. Today, a significant number of tools we use on a daily basis are powered by Machine Learning. Tools such as an internet search engine, maps, and recommendation engines rely on large swathes of data to attain the high accuracy that is expected of them. This has created an insatiable appetite for data as companies now have an incentive to collect data from their users. As more types of data are collected, the question of privacy becomes more important.

This thesis addresses some of the privacy concerns by proposing practically feasible protocols for machine learning, with privacy as the main focus. Secure Multiparty Computation protocols allow a set of parties to jointly compute on their data, without the parties having to exchange their data. We propose specialised versions of these protocols, tailor-made and optimised for the type of computations carried out in machine learning training and inference.

In the first work, we advance the field of secure inference by using a novel cryptographic object called an extended daBit. Extended daBits are theoretically interesting, and also lead to impressive gains in concrete applications. We give instantiations of highly efficient protocols for truncation, secure comparison, and non-linear functions (such as a rectified linear unit). Applying these techniques to applications such as biometric matching and convolutional neural networks shows a significant gain over prior work, making the framework practical.

Multiparty computation protocols most often support a static unchanging set of parties, which can be limiting for some applications. The second work, Le Mans, addresses this problem by proposing an efficient protocol that allows parties to leave and rejoin the computation, based on their resource availability. At the centre of this is what we call a universal preprocessing phase, which allows for parties to generate correlated randomness in a way that involves minimal communication in the online phase later. By taking advantage of pseudorandom correlation generators, we improve upon the state-of-the-art frameworks in the static setting. Finally, our online protocol demonstrates that supporting dynamic parties need not come at a huge cost in efficiency.

Resumé

Maskinlæring bliver mere og mere relevant med vidtrækkende indflydelse på tværs af industrier. Mange af de værktøjer vi bruger i vores dagligdag, er bygget på maskinlæring.

Værktøjer så som søgemaskiner, kort og anbefalingsalgoritmer, er afhængige af massive datamængder, for at opnå den høje præcision, som forventes af dem. Dette har skabt en stor appetit for data iblandt firmaer, som nu har stærke incitamenter til at indsamle data fra deres brugere. Spørgsmålet om privathed bliver vigtigere, som flere typer af data indsamles.

Denne afhandling afhjælper nogle privathedsudfordringer ved at præsentere praktisk anvendelige protokoller for maskinlæring, med privathed i centrum. Sikker Flerparts Beregning, eller MPC fra det engelske ”Secure Multiparty Computation”, tillader en gruppe parter at lave fælles beregninger uden at skulle udveksle deres inputdata direkte.

Vi fremlægger specialiserede versioner af disse protokoller, der er skræddersyet og optimeret efter typen af beregninger som er nødvendige under træning og brug af maskinlærings modeller.

I det første værk fremmer vi feltet af sikker inferens ved brug af et nyt kryptografisk objekt kaldt en forlænget daBit. Forlængede daBits er interessante fra et teoretisk synspunkt, og giver samtidigt også imponerende forbedringer i konkrete anvendelser. Vi giver instansieringer af meget effektive protokoller for trunkering, sikker sammenligning og ikke-lineære funktioner. Anvendelse af disse teknikker på biometrisk matchning of neurale netværk demonstrerer væsentlige forbedringer over tidligere arbejde, hvilket gør vores system praktisk anvendeligt.

MPC-protokoller understøtter oftest kun en fastlagt statisk gruppe parter, hvilket kan værre begrænsende i nogle sammenhæng. I det andet værk, kaldt Le Mans, imødekommer vi dette problem ved at præsentere en effektiv protokol, som tillader at parter forlader og senere tilslutter sig beregningen, baseret på ressourcerne de har tilgængelige. Kernen af denne protokol består af en universel præprocesseringsfase, som tillader parterne at generere korreleret tilfældighed ved brug af minimal kommunikation i en senere onlinefase. Ved at udnytte pesudo-tilfældige korrelationsgeneratorer forbedrer vi på de bedste kendte konstruktioner fra den statiske kontekst. Til sidst demonstrerer vores onlineprotokol at dynamiske parter kan understøttes uden den store effektivitetsomkostning.

Acknowledgments

There are a lot of people who were instrumental in making this thesis and my PhD happen. I would like to thank the entirety of Aarhus Crypto Group for nurturing me over the past three years. I believe that creating an environment where everyone is treated equally regardless of seniority and where no one is afraid to speak their mind, is not an easy thing to do. I was surprised how social the group was when I initially joined. We have lunch together every day, we play games, hangout in various social settings, go for sports, and lots of other things. It is always possible to find someone who is willing to have an impromptu conversation about research, even if it is not related to what they are working on. Cultivating such a positive environment is nothing short of an incredible accomplishment. What makes it even more impressive is how the group has kept the magic going, even as new people join in every year and the existing ones leave. I hope I can bring some of this spirit wherever I go in the future, and I would redo the last three years in a heartbeat (without the pandemic).

In short, Peter has been a terrific advisor. He always made time for me and supported me throughout the way. There have been times when I would go to him to have him explain something for the second or third time, and he was always understanding. I still admire the patience he has to entertain stupid simple questions. Even though he is a respected researcher, there not a hint of ego when talking to him. I believe almost every PhD student at some point feels like they are not doing enough, and they are disappointing their advisor. It is crucial to be able to talk about this with their advisor. I was fortunate to always be able to talk to Peter, and have him reassure me that I was doing great and had nothing to worry about.

Claudio was always there for me, checking up on me from time to time to make sure I was doing okay. I would like to thank him for instilling confidence in me during the early days of my PhD. I would like to thank Ashish Choudhury for encouraging me to pursue a PhD. I remember having conversations with him during my bachelor's, when he said I had the potential to pursue a PhD. I was not considering it at the time, and I am not sure what he saw in me, but I am glad he did. I would like to thank Yehuda Lindell and Assi Barak, who organised the Bar-Ilan Summer Internship program back in 2018, that gave me a taste of research. I would like to thank Benny Pinkas for hosting me at Bar-Ilan, mentoring me, and agreeing to be on my thesis committee. I would like to thank Adrià Gascón for also agreeing to be on the committee. I am indebted to all the people I have been fortunate to work with. In no particular order, thank you Daniel Escudero, Ajith Suresh, Carsten Baum, Nikolas Melissaris, Satrajit Ghosh, Marcel Keller, Mahak Pancholi, Mark Simkin, Lennart Braun, Matthew Jagielski, Nishat Koti, Arpita Patra, and Harsh Chaudhari.

I am grateful to all the friends that have stuck with me, and the ones I have made along the way. This journey would have been a lot more lonely without your support. Most importantly, I would like to thank my parents for their unconditional love and support and believing that I would be able to do this throughout the process.

Rahul Rachuri,
Aarhus, November 30, 2022.

Contents

Part I

Overview

Chapter 1

Introduction

Machine Learning is seeing widespread deployment in today's world. In a lot of ways it has become an indispensable tool. We are seeing it being used across a range of different industries. Use cases of Machine Learning range from building models that play some video games better, such as Go [SHM^{+}16] or DotA [OB^{+}19], and to more serious use cases such as cancer detection and self-driving vehicles. All these things have two things in common – they require huge amounts of data and computational power to train the models. DALL-E 2 [RDN^{+}22], a model that can create realistic images based on textual prompts, was trained using 250 million images. OpenAI Five, the bot trained to play DotA, was trained using 128,000 CPUs.

Though some of the core ideas for Machine Learning have been around for a couple of decades, it is only now that we are seeing the field take off with real world usage. One of the biggest factors for this is the increase in availability of cheap computing power. Smartphones have become immensely powerful. An iPhone today has over 100,000 times the computing power in its CPU and more than 1 million times the RAM of the Apollo Guidance Computer onboard the Apollo 11. Such computing power readily available in the palms of our hands opens the door for things that were not possible before, such as running Machine Learning models locally on our devices. There are a number of on-device Machine Learning APIs available on iOS and Android for things like image classification, speech recognition, and so on. These advances however, are not without downsides.

There are new ways of tracking and collection of data from users available today, in part because smartphones provide access to tap into a trove of data that was not accessible before. This report [TW19] by the New York Times shows how prevalent location tracking is with smartphones in the absence of proper protections. Even inside of the smartphone's OS, there are a lot of different ways users are tracked. Instagram and a host of other apps have recently been caught abusing in-app browsers to record every single tap made by the users [Kra22]. Allowing companies to collect and store this level of data creates new attack surfaces.

The type of data collected can range from something mundane and seemingly harmless, such as a search engine recording a user's search history or a keyboard on a smartphone recording keystrokes to improve its autocorrect engine, to collecting more sensitive data such as a healthcare provider recording a patient's history. It gets even more dangerous and privacy-invasive if these companies decide to share or combine their datasets, usually done under the pretext of improving their services or the user experience. Collaborating by sending data across different entities without any oversight opens the door for abuse. These entities could use the data for unintended purposes, or rogue actors in the system could misuse the data. This has led to an uproar against data abuse and gave rise to governmental regulations, such as the European Union General Data

Protection Regulation (GDPR). These regulations in their current state completely prohibit data sharing in certain contexts, thereby solving the problem of unfettered data sharing. But, there are certain valid purposes for which the entities might want to combine parts of their datasets. This could be something like hospitals collaborating on their datasets to train a new model for cancer detection.

The desire to combine datasets across a wide range of sources is not just to have huge datasets, it is beneficial that data used to train Machine Learning models has properties like variance in the dataset. If we take an example of someone training a model for facial recognition, ensuring that the dataset has a good representation of people around the world becomes paramount. We have seen evidence of what happens when datasets do not have this property [Con17]. In other words, a model trained on highly localised data usually does not perform very well when deployed in new settings around the world.

Training Machine Learning models has always been an arduous task. Even though computing power is getting cheaper, the computational cost of Machine Learning training is outpacing it. Modern Machine Learning models, such as the ResNet-50 [HZRS15], can have tens of millions of parameters and thousands of layers. Even with a lot of computing power, it can take days to train. Naturally, due to the increasing difficulty of training large models, we are seeing a new paradigm of Machine Learning training and prediction emerge that is referred to as Machine Learning as a Service (MLaaS). This paradigm introduces a few different ways to train models, by changing the traditional notion of the data owner also being the one that trains the model.

Outsourced Computation: In this variant, there is either a data owner that wants to train a model, or there is a model owner that wants to offer its model for prediction. The entity chooses resorts to hiring servers from cloud providers, such as Amazon AWS, Google Cloud, Microsoft Azure to do the computations for them. Even if we assume that the governmental regulations are a nonissue for the moment, which means the entity is allowed to store the data or the model on external servers, this model comes at a cost of privacy. The server hired for this task now becomes a single point of failure in the system, as anyone that gets access to the server gets access to the data or the model stored on it. Institutions like banks, with sensitive data about its customers, might not want to take such risks.

Community Outsourcing: Another model that has emerged due to data owners having limited computing resources, is a community outsourcing model. This model works well when there is a long, time-consuming computation to be carried out but the computation can be divided in several chunks, each of which is not very computationally intensive in and of itself. A data owner in this model, puts up these tasks on something like a public bulletin board, and anyone with free computational resources can join the network and carry out the tasks. This could be someone who leaves their laptop or smartphone plugged in overnight. The device joins the network, works on the tasks until the moment it is unplugged. At that point, it communicates the progress made back to the data owners, and leaves the network. The beauty of this model is that is democratises massive tasks such as analysing protein sequences, finding Mersenne primes, etc. Examples of this model include the Folding@home project, GPUGRID, Rosetta@home to name a few. This model is however, comes with a tradeoff. There is a significant amount of information leaked to the participants in the network, as the data corresponding to the task has to be sent to the participants. This limits the model from being used in other contexts with sensitive, personally identifiable information.

There have been many approaches that try to add privacy to the models described above, with different trade-offs. Differential privacy adds noise to the data such that even if one were to look at the data, they would not be able to infer much about any single data point. However, the trade-off here is that the result of the computation will have some error. Federated learning is a new model where there is one central server holding the model, and a set of participants with data. The server sends the model to all the participants, who then update the model by retraining it on their data. After a while everyone synchronises by sending their updates to the model to the central server. The server aggregates all the updates and sends the updated model to the participants, so that they can continue improving it.

Another way to tackle these problems is to use cryptography. Cryptography allows us to compute on our private data, without having to share the data itself. In this thesis, we will focus on providing solutions from a subdomain of cryptography, called secure multiparty computation. But given how multi-faceted the privacy problems are, we will see how there is no one-size-fits-all type of solution.

1.1 Cryptography

Cryptography was born out of a fundamental need to secure communications. End-to-End Encryption (E2EE) powers apps like Signal and WhatsApp, ensuring that no one except the intended recipient can read our communication. Encryption is also used to secure data at rest on hard drives, so that only having access to our devices does not give anyone access to the data contained with them. Over the years, as our devices got more powerful and there was more kinds of data being collected, there was a need to be able to *compute on private data*. Most widely deployed cryptography, Advanced Encryption Standard (AES) for instance, do not allow for computations on encrypted data out of the box. This led to new areas of cryptography, colloquially referred to as Modern Cryptography, which includes tools like Homomorphic Encryption (HE), Zero Knowledge Proofs (ZK), and Secure Multiparty Computation (MPC).

When data is encrypted, it is considered secure as no one can derive any meaningful information from it without having access to a special decryption key. When we want to compute on this data, we first have to decrypt it, perform the computation desired, and encrypt the data again. This might be fine when the data is on a laptop and the owner of the laptop is computing something locally on the data. If there is a service provider that stores the data of all its users, decrypting this data might not be desirable because the computation might take a long time and having the data decrypted for that long might be too much of a risk. Sometimes all we want to extract from the data is a particular property, in which case decrypting the entire database is an overkill. Homomorphic Encryption lets us avoid that by letting us compute on the encrypted data directly. We will obtain the result of the computation in an encrypted format, and we can decrypt only the result and not the entire database.

Despite being a solution to this problem on a technical level, homomorphic encryption is a long way from being practically efficient for computationally intensive tasks such as Machine Learning training. Currently it is orders of magnitude slower compared to computing on plaintext data (data that is not encrypted), making it a tough sell. Another approach to problem of computing on private data is secure multiparty computation, which will be the focus going forward.

Secure Multiparty Computation

Imagine we live in a world where there are trusted entities available. If two parties had some data that they wanted to compute on, but neither of them trusts the other one, they could send their data to this trust entity and ask it to run the computation. The entity would relay the result back to the parties, and delete their data from its storage. Sadly, we exist in the real world where just like unicorns and pixies, such entities do not exist. The area of MPC tries to provide such a functionality, given certain assumptions that we will get into.

More formally, MPC is a paradigm in cryptography that allows for multiple parties to compute on their data together, in a "secure" and "private" way. We will see later in this section how these terms are defined, and what security means in the context of MPC. At a fundamental level, it means that parties do not have to share their private data with anyone else in order to perform some computation. Parties only learn the final output of the computation, and nothing in between. MPC was first introduced by Andrew Yao in 1982 [Yao], with a tool called Garbled Circuits. Since then, research in MPC has been steadily growing. MPC protocols are typically generic in nature, meaning they allow for parties to compute any function on their data. In recent years however, there has been a tremendous amount of research in designing specialised protocols in MPC for particular use cases. Designing protocols geared towards a specific function, or class of functions, allows us to make certain design choices in the protocols that lead to much better efficiency.

The Adversary

In MPC protocols, security is defined with respect to an adversary, denoted by $\mathcal{A}$, in the system. Assume that there are n parties in a protocol. An adversary is someone that controls a fraction of the parties, referred to as corrupted parties, and tries to influence the protocol is some way. Depending on the number of corrupt parties, denoted by t, and what $\mathcal{A}$ can do with these parties, we have different types of adversaries.

Honest majority vs Dishonest majority: If $\mathcal{A}$ is allowed to corrupt up to a half of the parties in the system, we say that the protocol is secure in the honest majority setting. If $\mathcal{A}$ can corrupt all but one party in the system, that is referred to as a dishonest majority. A natural question to ask is why anyone would choose to build protocols in the honest majority setting as opposed to dishonest majority, since it is a weaker setting. Unsurprisingly though, honest majority protocols are simpler to design and are much faster compared to dishonest majority ones. Dishonest majority protocols are complicated and often have a lot of moving parts, so there is a lot of scope for someone to make a mistake while implementing them. In certain situations, the MPC protocol might only involve a few parties, which are well to known to everyone. Consider Amazon, Google and Meta trying to run an MPC protocol for something. There is an argument to be made that Amazon and Google would not collude to learn more about Meta's data. If they were to be caught, it will be irreparable damage to their reputation. So there are places where honest majority protocols make sense.

Semi-Honest vs Malicious: A semi-honest (passive) adversary is one that follows the protocol but tries to glean as much information from it as possible. A malicious (active) adversary on the other hand, can deviate from the protocol in any arbitrary way at any given time. $\mathcal{A}$ could choose to send a wrong message at any given step, it could choose to send no message at all, or it could send different messages to different parties. This type of adversary is significantly harder to deal with. If we were to go for real world deployment, assuming that the adversary might actively deviate from the protocol could be argued as a more realistic assumption in general. The exceptions to this are when the entities involved in the protocol have some trust in each other, but still cannot collaborate on their data directly due to reasons like regulations. The practicality of a passive adversary can be illustrated by the following example. Imagine there is a company that can collect data from its users, but is not allowed to because of regulatory reasons. One way to get around that could be that the company places its servers in different jurisdictions, and stores secret shares of the data on each of them. Since all the servers are owned by the same entity, there is little reason to worry about an active adversary in the system.

Static vs Dynamic Participants

Most of MPC protocols inherently assume that the set of parties that starts the computation will remain constant throughout. That is a reasonable assumption to make for most protocols, since the parties that start the computation are the ones with the data and the only ones interested in the output. Protocols usually also do not takes hours or days to run. However, for emerging models such as the community outsourcing model described earlier, this type of protocols are unfit.

In an MPC protocol designed for static participants, there is no fallback if one of them drops out. A drop out might occur due to any number of reasons, such as their system crashing. In such a situation, the protocols grinds to a halt because the remaining parties cannot proceed forward. There have been a certain class of MPC protocols designed to handle drop outs, but as we will see later, this is not enough for our use case. In our setting, we would like parties to be able to textitjoin mid-computation, and drop out whenever they want. There are extremely few MPC protocols that accommodate new parties joining midway through the protocol.

There have been folklore approaches to adding support for dynamic participants, mainly via committee-based MPC [Bra85] protocols. The general idea is to have a subset of parties perform the computation, that changes over the course of the computation. This however does not offer the flexibility offered in the plaintext model of community outsourcing. More recently, due to the advent of blockchains, dynamic protocols that offer more flexibility to participants are becoming increasingly relevant with works such as [CGMV18, BGG$^+$20, GHK$^+$21] leading the way.

Secret Sharing

The inputs of MPC protocols and intermediate values typically existed in a secret shared format. There are many different secret sharing schemes, the most common and relevant one for this thesis being additive secret sharing. In additive secret sharing, each party holds an additive share of the secret value, and the sum of all the parties' shares reconstructs the secret. We use the notation $[x]$ to denote a secret value x that is additively shared.

These shares are defined over a finite field or a ring. Typical choices for this is either a field $\mathbb{Z}_p$, where p is a prime, or $\mathbb{Z}_{2^k}$. So, an additive secret sharing of a value, say x, between four parties $P_1, \ldots, P_4$ would look as follows. Each party P_i holds a share s_i, and the relation between the shares would be $s = \sum_{i=1}^{4} s_i$, where s is the secret value.

Additive secret sharing is secure up to $t = n - 1$. That means even if $n - 1$ parties are corrupted, they still cannot learn the secret. Additive secret sharing is in the class of Linear Secret Sharing Schemes (LSSS). In these schemes, the addition operation comes for "free". To add two secret shared values, parties locally add the corresponding shares and get a secret sharing of the result. Multiplication is considered a non-linear operation, which means that parties have to communicate with each other to exchange some information that lets them compute the product of two secret shared values. Techniques to perform efficient multiplication of two secret shared values is a challenging problem, and has spawned a number of results tackling the issue with different trade-offs.

Authenticated Secret Sharing: In some situations, for instance when we are dealing with an active adversary in dishonest majority, a simple additive secret sharing of the secret does not suffice. Message authentication codes (MACs) are used to verify authenticity of the messages sent in the protocol. Each message in the protocol is associated with a tag by the sender, and the receiving parties use this tag to verify the integrity of the message. Consider the following MAC scheme, first introduced in [BDOZ11]. This is a pairwise MAC scheme, which means for a share that party P_i holds, it will also have a MAC with every other party P_j, denoted by M_j^i. P_j holds two values, a global key, denoted by Δ^j and a local key K_i^j. These values are related by the following equation:

$$M_j^i = [x] \cdot \Delta^j + K_i^j$$

The MACs are set up such that the global key and the local key are uniformly random, meaning that P_j does not learn any additional information about P_i's share. When parties want to reconstruct the secret x, each party broadcasts its share of x as before. In addition, they will send each of the pairwise MACs they hold to the corresponding party, who can verify that the MAC relation holds. This check fails with probability $1/|\mathbb{F}|$, where $\mathbb{F}$ is the field over which the MACs are defined. The proof for it is given in [BDOZ11].

The Preprocessing Paradigm

There are a lot of protocols that can take advantage of correlated randomness and result in a very fast online phase, when we actually do the computation on secret inputs. Because correlated randomness does not rely the secret inputs, we can split the MPC protocol into two phases – the preprocessing phase and the online phase. The preprocessing phase is used to generate the correlated randomness required for the protocol. Typically, generating correlated randomness is computationally heavy as it uses techniques such as cut-and-choose. The amortisation for them only kicks in when we generate a large amount of randomness, in the order of millions. Therefore, generating such amounts of randomness on the fly during the online phase will make the protocol expensive and slow.

One of the most popular choices for the type of correlated randomness used is called a Beaver triple [Bea92]. A Beaver triple is a set of values (a, b, c) that are additively shared among all the parties, related by the equation, $c = a \cdot b$. If we have such triples available in the protocol, we can obtain the product of $[x], [y]$ by having parties reconstruct the values $x - a$ and $y - b$ by broadcasting their shares. Then, using these values we can compute $[z]$ as $(x - a) \cdot (y - b) + (x - a) \cdot [y] + (y - b) \cdot [x] + [c]$. So in only 1 round, parties are able to get an additive secret sharing of the product, making this approach highly practical.

Efficiently generating triples has spawned a long line of results, [BLN$^+$15, FKOS15, FLNW17] to name a few, and continues to be an active area of research. The neat thing about Beaver triples is that the triples themselves are completely independent of the data used in the MPC computation. If we assume that we need triples in the order of say, one million, we could compute additive sharings of these triples before we even receive the data to do the MPC computation. Techniques mentioned above have communication cost that is linear in the number of triples to be generated. More recently, a new cryptographic object called pseudorandom correlation generators (PCGs) [BCG$^+$19b, BCGI18, BCG$^+$20, WYKW21a] have been proposed. PCGs allow two parties take short, correlated seeds, and expand them to generate a large amout of correlated randomness. PCG based techniques for the preprocessing have shown to achieve sublinear complexity in the communication, but also in storage, as parties only need to store a short seed until they start the online phase. In this thesis, we will show an extension to the two party PCG techniques to the n party case.

Simulated-based Security

For a few years after the first protocols in secure multiparty computation appeared, there was no formal framework to analyse security in a way that gave rigorous guarantees. Formalisation of security for MPC protocols came much later, in a few different flavours such as the stand-alone model [Can00] and the UC framework [Can01]. The UC framework is the more relevant one for the thesis, and the one we will be focusing on going forward.

The goal of the framework is to provide a formal way to prove that the adversary $\mathcal{A}$ does not learn anything from the intermediate message exchanged in the protocol, beyond what it already knows. In order to do this, we define an *ideal world*, and we will argue that the real world protocol is indistinguishable from the ideal world protocol. In the ideal world, there is an ideal functionality $\mathcal{F}$, a trusted third party that the parties can send their inputs to. The functionality computes the desired output and sends it to the parties. Defined as it is, the ideal world and the real world are trivially distinguishable to an adversary because in the real world, it will see messages exchanged between the parties whereas in the ideal world, parties only talk to the functionality. The communication pattern is clearly distinguishable. Therefore, we need to somehow simulate the real world protocol messages in the ideal world.

The simulator $\mathcal{S}$ sits in between the adversary and the ideal functionality in the ideal world. The goal of the simulator will be to simulate the protocol messages, *without* having access to the honest parties' inputs. It receives the protocol messages from the adversary, and responds on behalf of the honest parties. On the other side, the simulator talks to the functionality as if it were the adversarial party. In practice this means if the functionality expects an input from the adversary, the simulator extracts this from the adversary's messages, and forwards it to the functionality. The idea is that if the simulator is able to convince the adversary without relying on the inputs of the honest parties, it means that the protocol messages do not leak anything about the inputs.

There is one other entity that we have to worry about when doing the simulation, the environment $\mathcal{Z}$. The environment is trying to distinguish between the real world and the ideal world. We have already defined an adversary trying to do this, but that is not enough in one important way. If we only had the adversary from earlier, the simulator could employ a strategy in which the honest parties in the real world and the ideal world receive different outputs. The adversary does not have access to the honest parties' outputs, so it would not be able to distinguish between the two worlds in this case, even though it is not a perfect simulation. To address this issue, we use the environment, which tells the parties what inputs to use, and receives the outputs they receive in both worlds. In UC, we consider the environment and the adversary to be the same entity. So, at the end we want that the transcripts in both worlds, including the inputs and the outputs, follow the same distribution.

The reason UC is so powerful, is that it allows us to compose protocols together without much difficulty. Imagine there is a protocol Π', that internally runs a subprotocol Π. If Π, which realises a functionality $\mathcal{F}$, was proved secure in the UC framework, it means there is a simulator

$\mathcal{S}$ for it. This makes it simple, in that we can use the functionality $\mathcal{F}$ when describing Π'. And when doing the simulator for Π', we can simply run the simulator $\mathcal{S}$ to simulate the messages of Π.

Output Guarantees

Security in MPC is defined by comparing the real world and the ideal world. The guarantee that the parties in the real world have is that, the protocol they are running is equivalent to send their secret inputs to the ideal functionality, where only the output is revealed. However, when we say the ideal functionality reveals only the output, that can happen is multiple ways.

Security with Abort: In this case, the adversary has the power to abort the protocol at any given point, including in the output phase of the protocol. This means the adversary could wait until it receives the output, and then cause all the honest parties in the system to abort. In some cases, the adversary can choose a subset of the honest parties to receive the output and the rest to abort. This variant is referred to as Security with Selective Abort.

Fairness: Fairness improves upon Security with Abort. The adversary in this system can still cause aborts, but if the corrupt parties learn the output, we are guaranteed that all the honest parties also learn the output.

Guaranteed Output Delivery (GOD): This property, sometimes also called Robustness, is the strongest guarantee we have, which guarantees that no matter how the adversary deviates from the protocol, the honest parties will learn the output of the protocol. Robustness is the hardest property to realise and in some cases is not possible at all.

Identfiable Abort: Identifiable Abort is a relatively new notion of security, first appearing in [CL17, IOZ14]. It is similar to abort in the sense that the adversary can abort the protocol at any point, but with the caveat that the adversary must point to a party, or a set of parties, as the corrupt ones before aborting. Although not as strong as GOD, such a notion might be useful in settings where there are only a few parties controlled by the adversary. If the adversary cheats and gets caught, these parties will be kicked out of the system and the remaining parties will restart the computation.

These notions are more relevant in the case when the adversary is actively corrupt. Because if the adversary is passive, we know that it will not deviate from the steps of the protocol, meaning it will never cause an abort and the honest parties will always learn the output. So, we trivially get GOD in the passive setting. However, for some corruption thresholds these guarantees are shown to be impossible in [Cle86, GMW87, RB89].

Efficiency of MPC Protocols

There are a number of different ways to measure the efficiency of an MPC protocol. The two most common metrics are the amount of communication and the number of rounds. Communication is measured by counting the number of bits sent by each party throughout the course of the protocol. Protocols optimised to minimise communication are referred to as high throughput protocols. On the other side of the spectrum, we have low-latency, or round optimised protocols. These protocols might incur more cost when it comes to bandwidth, but they minimise the number of rounds of communication between the parties. Such protocols make sense when the round trip time (RTT) between parties is high, or the connection is unstable so we would prefer not communicating over multiple rounds. There is also a measure of storage, which counts how much data each party has to retain throughout the computation. Though storage is cheaper to come by than bandwidth or low-latency, when it comes to running MPC protocols on Internet of Things (IoT) devices which might be storage limited, it becomes an important to metric to optimise for.

There are couple of other choices to measure efficiency of protocols by, like run time and monetary cost. Run time simply measures the clock time from the start of the protocol to the end. Monetary cost is newer metric, that is gaining relevance as outsourcing servers is becoming more popular. Monetary cost is how much it would cost to the protocol should one hire the appropriate servers for it.

Run time and monetary cost are especially important for long running computations such as machine learning. When one hires servers from cloud providers, they pay for it based on multiple factors, such as bandwidth used, computational power of the servers, and the up time of the server. If we were to take a communication efficient protocol that takes a long time to run, in an attempt to reduce the bandwidth used, it would not help as the monetary cost would be high because of the long uptime of the servers. Therefore, optimising for low monetary cost requires striking a balance between all the metrics we described.

1.2 Private Machine Learning

Private Machine Learning involves techniques for performing secure machine learning training and secure inference. A typical scenario for machine learning training is when there is a set of parties, each with a dataset. They want to collaborate on their data to jointly train a model. Even after the model has been trained, they do not want to reveal the entire model to any party. The model should exist in some "secret" state, but should support for prediction queries from the parties. When it comes to secure inference, imagine there is an entity that possesses a trained model. It wants to open this model up for queries from clients, and charge them for query access. The entity wants to hire an external server for this purpose. In this scenario, there are two kinds of privacy concerns. On the model owner's side, it does not want the hired server to learn the model, because it is proprietary data. On the client's side, it does not want to expose the query to the server or the model owner, as it might contain sensitive information. The client wants to receive the output of the prediction without the server learning any information about the query. Both these tasks present challenges that are unique, and require designing protocols specific to them.

The Problem with Floating Point Arithmetic: Data point and intermediate values in machine learning are real values. While not a problem in the plaintext setting, it becomes a significant roadblock in the MPC domain. As mentioned earlier, values in MPC are typically defined over a finite field, where there is no natural way represent a fractional value. One approach could be to drop the values after the decimal point and train using only the integer part. This creates a drop in accuracy compared to the plaintext model trained on fractional values, that is too high to tolerate.

The most popular approach, Fixed Point Arithmetic (FPA) to get around this issue has been to allocate a set of lower order bits of a finite field to represent the fractional part of the values, and use the higher order bits for the integer part. For instance, if we were operating with a 64-bit field, the lower 13 bits are typically reserved for the fractional part, 1 bit for the decimal point, and the rest are used for the integer part.

Addition of two FPA numbers is straightforward, we can add the shares of these values locally. Multiplication however, poses new problems. The product of two numbers in their FPA representation moves the decimal point higher. Going by the example from above, the product would now have 26 bits of precision. We need to truncate the product by 13 bits before proceeding forward. There are multiple approaches to truncation, involving different trade-offs in the accuracy. The most common and communication efficient approach is called probabilistic truncation, which truncates the result but it might be off by a very small amount.

Mixed-Circuit Computation: As mentioned earlier, typical choice for the mode of computation in MPC is either a ring of the form $\mathbb{Z}_{2^k}$ or $\mathbb{F}_p$, where p is a prime. A function we want to compute over these domains is expressed as a "circuit" in MPC literature. A circuit is comprised of the following types of gates: input gates, addition gates, multiplication gates, and output gates. The input and output gates are used to plug in inputs to the circuit and receive the output, respectively.

An arithmetic circuit defined over a field $\mathbb{F}_p$, where $p = 2$, along with the operations XOR and AND, is called a binary circuit. Arithmetic and binary circuits come with different trade-offs. Arithmetic circuits are more efficient when it comes to operations like multiplication, which are very expensive to do with binary circuit. On the other hand, binary circuits are more suitable for operations like bit extraction, where we want to extract a bit from a given position of an l bit value, or for secure comparison of two values.

Switching between different modes of computation mid-way through an MPC protocol can be extremely useful in some cases. If there were a function where multiplications and comparisons alternate, computing this function in either domain will not be particularly efficient. Functions evaluated as part of machine learning, such as neural network, often resemble this structure. Ideally, we would like to be able to perform the multiplications in the arithmetic domain, but somehow translate these values from the given field $\mathbb{F}_p$ to $\mathbb{F}_2$. It would allow us to evaluate operations like comparisons in the binary world, maintaining efficiency. This insight led to the development of what are referred to as circuit conversion techniques. There has been a line of works in the recent years that have developed highly efficient ways to switch between different computational domains in different settings of MPC.

Non-Linear Functions: Most common operations in machine learning can be put into two categories. The first contains operations such as matrix multiplications, convolutions, etc. We refer to these as linear operations, and they can be realised in MPC using arithmetic circuits without much difficulty. The second set of operations consists of Rectified Linear Unit (ReLU), sigmoid, softmax etc. These activations functions are a common occurrence, and they are not easy to realise in MPC. If we take the example of softmax, it is defined as follows for a vector of values $\boldsymbol{x}$ of length K:

$$\mathsf{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

Computing the exponent in this equation, and the division are non-trivial in MPC. Due to this, the most common approach towards realising such non-linear functions in MPC is to come up with an MPC-friendly approximation for the functions. It is a common approach to use an approximation or a simplified version of these functions, where the difficult operations are replaced with MPC-friendly alternatives, as shown in [MZ17, MR18, WGC19]. These approximations come with different trade-offs. For example, [MZ17] replaces the exponentiations with ReLU operations and uses a division circuit based on garbled circuits, making it more MPC-friendly. They show that the accuracy drop for training MNIST with the approximation of softmax is only about 1%. However, it was shown in [KS20] that there is a considerable drop in accuracy when this technique is applied to multilayer perceptrons. As an improvement, [KS22] proposed a different method of computing softmax, where the focus was on improving the accuracy. Their approach involves an efficient algorithm for computing exponentiations in MPC, and they show that they are within 0.2% in terms of accuracy delta compared to the plaintext model for an MNIST classifier.

Chapter 2

Approaches to Private Machine Learning

This chapter outlines works I have been a part of in the three years of my PhD. The works can broadly be categorised under practical MPC protocols, with a focus on optimising for machine learning operations. As we will see in this chapter, each of these works attacks the problems of private machine learning in a unique way, bringing privacy-preserving machine learning closer to reality.

- [EGK$^+$20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO 2020, Part II*, LNCS. Springer, Heidelberg, August 2020

- [RS22] Rahul Rachuri and Peter Scholl. Le mans: Dynamic and fluid MPC for dishonest majority. In *CRYPTO 2022, Part I*, LNCS. Springer, Heidelberg, August 2022

2.1 edaBits: Mixed-Circuit Conversions for MPC

There has been a long line of works [DSZ15, MR18, MZ17, HKS$^+$10, WGC19] in MPC proposing conversions between the different domains of arithmetic, binary, and garbled/Yao circuits. Most of these works are set in the honest majority setting. There have been few works in the dishonest majority setting for more than two parties, with one of the most recent ones being [RW19]. This work introduces a new cryptographic object called daBits (doubly-authenticated bits). A daBit is a secret shared random bit, that is shared over both the arithmetic and binary domains. daBits are extremely useful for conversions between the arithmetic and binary domains, and versatile, which means that they could be used in any corruption setting. There are a few different ways to generate daBits, with different trade-offs. Despite their versatility and usefulness, daBits are limited when it comes to real world applications. Taking the example of Machine Learning, the values we work with most of the time are not single bit integers, but integers over a much larger domain. To convert an integer over this domain using daBits, we would have to use multiple daBits. Due to the cost of generating daBits, the efficiency when converting a value secret shared over a large domain comes out to a factor of three at best.

In this work, we introduce a new object called an extended daBit (edaBit), that can be used to convert between different domains. An edaBit is a set of m random bits, $r_1, \ldots, r_m$, secret shared over the binary domain, and their corresponding value $r = \sum_{i=1}^{m} r_i \cdot 2^i$ shared over the arithmetic domain. A large part of the paper is dedicated to showing how to efficiently preprocess edaBits in the dishonest majority setting with active security. Our starting point is the cut-and-choose protocol, which was originally introduced to produce a large number of multiplication triples at a low cost. Adapting the method to edaBits turns out to not be straightforward. The cut-and-choose procedure involves taking a large batch of triples, and placing them in buckets of a particular size. Within each bucket, one triple is selected and a check is performed with every other triple. At the end, we are guaraanteed that this triple has not been tampered with, except with negligible probability. Trying to follow the same template with edaBits poses a problem, which is that the check that needs to be performed to check two edaBits is different from the check with the triples in one important way. The triples have a linear relationship as opposed to edaBits, meaning that we require additional tool(s) to check two edaBits. We need to use TinyOT triples [NNOB12, FKOS15] to do the pairwise checks. The naive approach would be to first run a cut-and-choose to produce secure TinyOT triples, using prior approaches, and use them for the cut-and-choose for edaBits.

In an effort to optimise this further, we forgo generating secure authenticated TinyOT triples. Instead we let the party that secret shares the triples, tamper with them. So we get authenticated triples, but they are not checked for correctness. On the face of it, this seems like the adversary gets too much power, and can cheat in the cut-and-choose procedure to make it output faulty edaBits. For instance, if a tampered TinyOT triple was used for the check involving a good edaBit and a tampered one, this might cause the check to pass, depending on how the triple and the edaBit were tampered. We show that, if the circuit has what we call *weak additive tamper resilience*, it significantly hampers the adversary's ability to pass the checks.

Using edaBits, we provide protocols for several primitives such as probabilistic and deterministic truncation, secure comparison, and equality test. In addition, we show that our constructions work with both signed and unsigned data types. edaBits provide the most gain in the dishonest majority setting with active security, where the improvement is up to a factor of 25 over daBits for secure comparison. We also benchmark applications such as biometric matching against the ABY [DSZ15] and HyCC [BDK$^+$18] frameworks, and convolutional neural networks against the state-of-the-art framework of [DEK20]. For neural network inference, we observe an improvement of up to a factor of 6 in both communication and run time.

2.2 Le Mans: MPC for Dynamic Participants

MPC protocols are typically designed for a static set of participants. If there is a party that wants to leave the computation for a brief period of time, even as brief as one round of the computation, this is not allowed. The protocol will abort, and the remaining parties will restart the computation. When the party comes back at a later point, it will also not have any way to rejoin the computation. This is problematic for a host of reasons. If we are running a computation that may take several days, there may be good reasons why a party might want to leave the computation midway, before rejoining at a later point.

Committee-based MPC protocols [Bra85, GSY21, MZW$^+$19] have existed for some time, but the goal of these protocols is not to give parties the flexibility to leave and rejoin the computation. One could try and bootstrap these protocols to allow for a dynamically evolving set of parties in the computation, but the resulting protocols are messy and impractical. This work takes the model of Fluid MPC [CGG$^+$21], and adapts it for the dishonest majority setting with active security.

The main challenge in designing a fluid protocol for the dishonest majority is that most of the efficient protocols in dishonest majority involve the use of correlated randomness, typically multiplication triples. The Fluid MPC model does not have to use a preprocessing phase, since it is designed for the honest majority case. Even if we make the concession of using a preprocessing phase for the dishonest variant, there are still challenges in using triples.

There are efficient ways to preprocess multiplication triples [DPSZ12, DKL$^+$13, FKOS15], but they do not provide a good solution for fluid protocols. It is because a triple is an additively secret shared set of values $[a], [b], [c]$, where $c = a \cdot b$. Imagine we designed the protocol by having all the parties that *might* participate in the online phase of the protocol, join the preprocessing to generate triples. This immediately restricts the online set of parties to everyone from the preprocessing because if we remove even one party's shares of the triples, the remaining shares do not add up and maintain the same relation between the a, b, and c components.

To work around this problem, we introduce a *universal preprocessing phase*. The goal of the universal preprocessing phase is to let n parties generate correlated randomness, such that any subset of the parties can run the online phase, by simply ignoring values corresponding to the missing parties. Instead of generating shares of the standard multiplication triples, the parties generate *partial triples*. The a and b values in the triple are additively shared as before, but for the c component, we only distribute shares of the cross terms $a^i \cdot b^j$, to P_i, P_j, where a^i is the share held by P_i and b^j is the share held by P_j. If P_j drops out of the computation, all the parties ignore the shares corresponding to P_j and sum up the remaining shares to a fresh looking share of c.

Using the universal preprocessing, we give two online phase protocols, for two different scenarios. The first one is called Dynamic SPDZ, which is designed to tolerate one crash that might occur after the preprocessing. In the traditional SPDZ protocol, if a party crashes after the preprocessing, there is no way to proceed except for the remaining parties to redo the preprocessing. Since this is usually computationally intensive and might take a long time, it is not ideal to have to redo it. In Dynamic SPDZ, parties ignore the shares corresponding to the missing party and proceed with the online phase. Dynamic SPDZ costs only 8 elements on top of the traditional SDPZ protocols such as [DPSZ12, KPR18, KOS16]. In addition, universal preprocessing is significantly cheaper than the traditional variant both in terms of communication and storage complexity.

Designing a *maximally fluid* online protocol, where the expectation is that parties leave and rejoin the protocol in every single round, has an additional challenge. When a party is leaving the protocol after a given round, it reshares the shares it holds to parties that are going to continue with the computation. This works for additively shared values, but does not work with authenticated shares. An authenticated share has a MAC, and the MAC key is the sum of all the keys of the parties in the protocol. So when a party leaves or joins the protocol, doing a resharing of the MACs will not work as the MAC key of the parties of the subsequent round will be different. We propose a very simple and efficient way to work around this problem, with a protocol called Key-Switch. It allows for parties in a given round to reshare values such that the MAC key is adjusted to match the MAC key of the parties in the subsequent round, with just one set of messages.

2.3 Additional Research - Actively Secure 4PC

Apart from these works, I have also been a part of some works which are summarised below.

- [RS19] Rahul Rachuri and Ajith Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. Cryptology ePrint Archive, Report 2019/1315, 2019. `https://eprint.iacr.org/2019/1315`. Published in NDSS 2020.

- [KPRS21] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. Tetrad: Actively secure 4PC for secure training and inference. Cryptology ePrint Archive, Report 2021/755, 2021. `https://eprint.iacr.org/2021/755`. Published in NDSS 2022.

MPC for a small number of parties has taken off in the recent years, with works such as [AFL$^+$16, FLNW17, MRZ15] proposing highly efficient protocols. These works deal with the specific setting of four parties, with an honest majority and active security. The three parties and one corruption setting has seen a lot of traction in the recent years [MR18, WGC19]. The thing that is common between all these works is that the passively secure versions of the protocols are much faster than the active ones. Although it is expected that the actively secure protocols will be slower, the slowdown in these protocols is in particular due to the cost of operations like truncation, dot product. The following works started out with the goal of finding out if the performance delta between the three party and four party settings is big enough to justify assuming the presence of an additional honest party.

Trident: Trident answered the question affirmatively, with up to 187x gain in the training phase, and 158x gain in secure inference compared to ABY3 [MR18], which at the time was the state-of-the-art framework for three parties. Trident is a four-party framework with actvie security, and is optimised to provide a fast online phase by making use of a preprocessing phase. At the heart of Trident is a variant of the replicated secret sharing scheme, that enables demarcation of roles to the parties. The multiplication protocol of Trident improved upon the state-of-the-art protocol of [GRW18], by trading off reducing the online cost by one element, in exchange for increasing the preprocessing cost by one. Therefore, the online cost is 3 elements, 1 lower than [GRW18]. We also showed how to increase the security level to fairness, without any additional cost to the multiplication protocol.

Along with the multiplication protocol, Trident also has support for conversions between different modes of computation. As a result of having one additional honest party, we were able to eliminate the use of the Ripple Carry Adder circuit in a few but important places where ABY3 required them, thus making it more efficient. For instance, our binary to arithmetic (B2A) conversion protocol requires only 1 round, compared to the $1 + \log l$ rounds of ABY3. When it comes to activation functions like ReLU and Sigmoid, ABY3 requires rounds proportional to the size of the underlying ring, whereas Trident has a constant cost of 4 and 5 rounds respectively.

One interesting feature of the design of Trident is that, it allows for one of the four parties to be turned off for a majority of the online phase. The fourth party will only need to be online for the input sharing and output reconstruction phases, saving in terms of monetary cost compared to [GRW18].

Tetrad: Tetrad is similar to Trident in that it is a four-party framework also set in the preprocessing paradigm, and the focus is on providing a fast online phase. Tetrad directly improved upon Trident in many ways. To start with, Tetrad has a different variant of the four-party replicated secret sharing scheme used in Trident. On the back of this secret sharing scheme, the cost of the multiplication protocol is reduced from 6 ring elements to 5 in Tetrad. In addition, we show how to obtain robustness for the same amortised cost over the fair multiplication protocol, something that was not possible with Trident. The robust variant improves the communication complexity compared to the state-of-the-art robust protocols of [KPPS21, DEK21].

Even though Tetrad is set in the preprocessing paradigm, it is not limited to it. Typically, if we were to try and convert protocols that have a preprocessing phase to online-only protocols, we would have no choice but to run the preprocessing and the online phases sequentially. However, this is not the case with Tetrad. We can parallelise operations such that the round complexity of the online-only version of Tetrad is exactly the same as that of the online phase from before, adding to the versatility of the framework.

We show how to achieve dot product and probabilistic truncation without any additional overhead over a multiplication for the first time with fairness and robustness. Given the prominence of these operations in machine learning applications, this has a substantial impact on the performance. It is also fairly common to have to multiply 3 or 4 inputs at once in these applications. The standard approach is to use the tree-based technique, but this incurs an additional round compared to the 2 input multiplication. Since the goal of Tetrad is to provide an efficient online phase, we propose protocols for 3 and 4 input multiplications maintaining the same round complexity as the 2 input one, at the cost of some additional communication in the preprocessing.

We often want to convert to the garbled domain in machine learning applications only for specific non-linear operations, and switch back to the arithmetic world after, as non-linear are usually followed by a multiplication. In Tetrad, on top of improving techniques for conversions compared to Trident, we also propose a new approach to them called end-to-end conversions. An end-to-end conversion of the form A-G-A takes as input shares in the arithmetic domain, evaluates the garbled circuit, and gives the final output shares in the arithmetic domain. To facilitate end-to-end conversions, we use a garbled circuit protocol that is built specifically for this purpose.

Monetary cost is also one of the tent-pole features of Tetrad. We take the idea of turning off one of the parties of Trident a step further, by allowing two parties to be off for most of the online phase. The impact of this is pronounced with long running computations, where we observe up to a factor 6 improvement over the cost of Trident. In terms of absolute performance, Tetrad is up to 4 times faster in runtime over Trident. The primary reason for the runtime improvement comes from the reduction in the number of rounds required through the use of multi-input gates.

Part II

Publications

Chapter 3

edaBits: Mixed-Circuit Conversions for MPC

The contents of this chapter have been taken from the paper [EGK$^+$20]. The only modifications made are to move the appendix content into the main body for better readability.

3.1 Introduction

Secure multi-party computation, or MPC, allows a set of parties to compute some function f on private data, in such a way that the parties do not learn anything about the actual inputs to f, beyond what could be computed given the result. MPC can be used in a wide range of applications, such as private statistical analysis, machine learning, secure auctions and more.

MPC protocols can vary widely depending on the adversary model that is considered. For example, protocols in the *honest majority* setting are only secure as long as fewer than half of the parties are corrupt and colluding, whilst protocols secure against a *dishonest majority* allow all-but-one of the parties to be corrupt. Another important distinction is whether the adversary is assumed to be *semi-honest*, that is, they will always follow the instructions of the protocol, or *malicious*, and can deviate arbitrarily. [ZLWL21]

The mathematical structure underpinning secure computation usually requires to fix what we call a computation domain. The most common examples of such domains are computation modulo a large number (prime or power of two) or binary circuits (computation modulo two). In terms of cost, the former is more favorable to integer computation such as addition and multiplication while the latter is preferable for highly non-linear functions such as comparisons.

Applications often feature both linear and non-linear functionality. For example, convolution layers in deep learning consist of dot products followed by a non-linear activation function. It is therefore desirable to convert between an arithmetic computation domain and binary circuits. This has led to a line of works exploring this possibility, starting with the ABY framework [DSZ15] (Arithmetic-Boolean-Yao) in the two-party setting with semi-honest security. Other works have extended this to the setting of three parties with an honest majority [MR18, ABF$^+$18], dishonest majority with malicious security [RW19], as well as creating compilers that automatically decide which parts of a program should done in the binary or arithmetic domain [BDK$^+$18, IMZ19, CGR$^+$19].

A particular technique that is relevant for us is so-called *daBits* [RW19] (doubly-authenticated bits), which are random secret bits that are generated simultaneously in both the arithmetic and binary domains. These can be used for binary/arithmetic conversions in MPC protocols with any corruption setting, and have in particular been used with the SPDZ protocol [DPSZ12], which provides malicious security in the dishonest majority setting. Later works have given more efficient ways of generating daBits [AOR+19, RST+19, BST20], both with SPDZ and in the honest majority setting.

Another recent work uses function secret sharing [BGI15] for binary/arithmetic conversions and other operations such as comparison [BGI19]. This approach leads to a fast online phase with just one round of interaction and optimal communication complexity. However, it requires either a trusted setup, or an expensive preprocessing phase which has not been shown to be practical for malicious adversaries.

Limitations of daBits. Using daBits, it is relatively straightforward to convert between two computation domains. However, we found that in application-oriented settings the benefit of daBits alone is relatively limited. More concretely, if daBits are used to compute a comparison between two numbers that are secret-shared in $\mathbb{Z}_M$, for large arithmetic modulus M, the improvement is a factor of three at best. The reason for this is that the cost of creating the required daBits comes quite close to computing the comparison entirely in $\mathbb{Z}_M$. This limitation seems to be inherent with any approach based on daBits, since a daBit requires generating a random shared bit in $\mathbb{Z}_M$. The only known way of doing this with malicious security require first performing a multiplication (or squaring) in $\mathbb{Z}_M$ on a secret value [DFK+06, DEF+19]. However, secret multiplication is an expensive operation in MPC, and doing this for every daBit gets costly.

Our Contributions

In this paper, we present a new approach to converting between binary and arithmetic representations in MPC. Our method is general, and can be applied to a wide range of corruption settings, but seems particularly well-suited to the case of dishonest majority with malicious security such as SPDZ [DPSZ12, DKL+13], over the arithmetic domain $\mathbb{Z}_p$ for large prime p, or the ring $\mathbb{Z}_{2^k}$ [CDE+18]. Unlike previous works, we do not generate daBits, but instead create what we call *extended daBits* (edaBits), which avoid the limitations above. These allow conversions between arithmetic and binary domains, but can also be used directly for certain non-linear functions such as truncations and comparisons. We found that, for two- and three-party computation, edaBits allow to reduce the communication cost by up to two orders of magnitude and the wall clock time by up to a factor of 50 while both the inputs as well as the output are secret-shared in an arithmetic domain.

Below we highlight some more details of our contribution.

Extended daBits.

An edaBit consists of a set of m random bits $(r_{m-1}, \dots, r_0)$, secret-shared in the binary domain, together with the value $r = \sum_{i=0}^{m-1} r_i 2^i$ shared in the arithmetic domain. We denote these sharings by $[r_{m-1}]_2, \dots, [r_0]_2$ and $[r]_M$, for arithmetic modulus M. Note that a daBit is simply an edaBit of length $m = 1$, and m daBits can be easily converted into an edaBit with a linear combination of the arithmetic shares. We show that this is wasteful, however, and edaBits can in general be produced much more efficiently than m daBits, for values of m used in practice.

Efficient malicious generation of edaBits.

Let us first consider a simple approach with semi-honest security. If there are n parties, we have each party locally sample a value $r^i \in \mathbb{Z}_M$, then secret-shares r^i in the arithmetic domain, and the bits of r^i in the binary domain. We refer to these sharings as a *private* edaBit known to P_i. The parties can combine these by computing $\sum_i r^i$ in the arithmetic domain, and executing $n-1$ protocols for addition in the binary domain, with a cost $O(nm)$ AND gates. Compared with using daBits, which costs $O(m)$ secret multiplications in $\mathbb{Z}_M$, this is much cheaper if n is not too large, by the simple fact that AND is a cheaper operation than multiplication in MPC.

To extend this naive approach to the malicious setting, we need a way to somehow verify that a set of edaBits was generated correctly. Firstly, we extend the underlying secret-sharing scheme to one that enforces correct computations on the underlying shares. This can be done, for instance, using authenticated secret-sharing with information-theoretic MACs as in SPDZ [DPSZ12]. Secondly, we use a cut-and-choose procedure to check that a large batch of edaBits are correct. This method is inspired by previous techniques for checking multiplication triples in MPC [BLN$^+$15, FKOS15, FLNW17]. However, the case of edaBits is much more challenging to do efficiently, due to the highly non-linear relation between sharings in different domains, compared with the simple multiplicative property of triples (shares of (a, b, c) where $c = ab$).

Cut-and-choose approach. Our cut-and-choose procedure begins as in the semi-honest case, with each party P_i sampling and inputting a large batch of private edaBits of the form $(r^i_{m-1}, \dots, r^i_0), r^i$. We then run a verification step on P_i's private edaBits, which begins by randomly picking a small subset of the edaBits to be opened and checked for correctness. Then, the remaining edaBits are shuffled and put into buckets of fixed size B. The first edaBit in each bucket is paired off with every other edaBit in the bucket, and we run a checking procedure on each of these pairs. To check a pair of edaBits r, s, the parties can compute $r + s$ in both the arithmetic and binary domains, and check these open to the same value. If all checks pass, then the parties take the first private edaBit from every bucket, and add this to all the other parties' private edaBits, created in the same way, to obtain secret-shared edaBits. Note that to pass a single check, the adversary must have corrupted both r and s so that they cancel each other out; therefore, the only way to successfully cheat is if every bucket with a corrupted edaBit contains *only* corrupted edaBits. By carefully choosing parameters, we can ensure that it is very unlikely the adversary manages to do this. For example, with 40-bit statistical security, from the analysis of [FLNW17], we could use bucket size $B = 3$ when generating more than a million sets of edaBits.

While the above method works, it incurs considerable overhead compared with similar cut-and-choose techniques used for multiplication triples. This is because in every pairwise check within a bucket, the parties have to perform an addition of binary-shared values, which requires a circuit with $O(m)$ AND gates. Each of these AND gates consumes an authenticated multiplication triple over $\mathbb{Z}_2$, and generating these triples themselves requires additional layers of cut-and-choose and verification machinery, when using efficient protocols based on oblivious transfer [NNOB12, FKOS15, WRK17b].

To reduce this cost, our first optimization is as follows. Recall that the check procedure within each bucket is done on a pair of *private* values known to one party, and not secret-shares. This means that when evaluating the addition circuit, it suffices to use *private* multiplication triples, which are authenticated triples where the secret values are known to party P_i. These are much cheaper to generate than fully-fledged secret-shared triples, although still require a verification procedure based on cut-and-choose. To further reduce costs, we propose a second, more significant optimization.

Cut-and-choose with faulty check circuits. Instead of using private triples that have been checked separately, we propose to use *faulty private triples*, that is, authenticated triples that are not guaranteed to be correct. This immediately raises the question, how can the checking procedure be useful, if the verification mechanism itself is faulty? The hope is that if we randomly shuffle the set of triples, it may still be hard for an adversary who corrupts them to ensure that any incorrect edaBits are canceled out in the right way by the faulty check circuit, whilst any correct edaBits still pass unscathed. Proving this, however, is challenging. In fact, it seems to inherently rely on the *structure* of the binary circuit that computes the check function. For instance, if a faulty circuit can cause a check between a good and a bad edaBit to pass, and the same circuit also causes a check between two good edaBits to pass, for some carefully chosen inputs, then this type of cheating can help the adversary.

To rule this out, we consider circuits with a property we call *weak additive tamper-resilience*, meaning that for any tampering that flips some subset of AND gate outputs, the tampered circuit is either incorrect for every possible input, or it is correct for all inputs. This notion essentially rules out input-dependent failures from faulty multiplication triples, which avoids the above attack and allows us to simplify the analysis.

Weak additive tamper-resilience is implied by previous notions of circuits secure against additive attacks [GIP$^+$14], however, these constructions are not practical over $\mathbb{F}_2$. Fortunately, we show that the standard ripple-carry adder circuit satisfies our notion, and suffices for creating edaBits in $\mathbb{Z}_{2^k}$. However, the circuit for binary addition modulo a prime, which requires an extra conditional subtraction, does not satisfy this. Instead, we adapt the circuit over the integers to use in our protocol modulo p, which allows us to generate length-m edaBits for any $m < \log p$; this turns out to be sufficient for most applications.

With this property, we can show that introducing faulty triples does not help an adversary to pass the check, so we can choose the same cut-and-choose parameters as previous works on triple generation, while saving significantly in the cost of generating our triples used in verification. The bulk of our technical contribution is in analysing this cut-and-choose technique.

Silent OT-friendly.

Another benefit of our approach is that we can take advantage of recent advances in oblivious transfer (OT) extension techniques, which allow to create a large number of random, or correlated, OTs, with very little interaction [BCG$^+$19b]. In practice, the communication cost when using this "silent OT" method can be more than 100x less than OT extension based on previous techniques [IKNP03], with a modest increase in computation [BCG$^+$19a]. In settings where bandwidth is expensive, this suits our protocol well, since we mainly use MPC operations in $\mathbb{F}_2$ to create edaBits, and these are best done with OT-based techniques. This reduces the communication of our edaBits protocol by an $O(\lambda)$ factor, in practice cutting communication by 50–100x, although we have not yet implemented this optimization.

Note that it does not seem possible to exploit silent OT with previous daBit generation methods such as by Aly et al. [AOR$^+$19]. This is due to the limitation mentioned previously that these require a large number of random bits shared in $\mathbb{Z}_p$, which we do not know how to create efficiently using OT.

Applications: improved conversions and primitives.

edaBits can be used in a natural way to convert between binary and arithmetic domains, where each conversion of an m-bit value uses one edaBit of length m, and a single m-bit addition circuit. (In the mod-p case, we also need one "classic" daBit per conversion, to handle a carry computation.) However, for many primitives such as secure comparison, equality test and truncation, a better approach is to exploit the edaBits to perform the operation without doing an explicit conversion. In the $\mathbb{Z}_{2^k}$ case, a similar approach was used previously when combining the SPDZ2k protocol with daBit-style conversions [DEF$^+$19]. We adapt these techniques to work with edaBits, in both $\mathbb{Z}_{2^k}$ and $\mathbb{Z}_p$. As an additional contribution, more at the engineering level, we take great care in all our constructions to ensure they work for both signed and unsigned data types. This was not done by previous truncation protocols in $\mathbb{Z}_{2^k}$ based on SPDZ [DEF$^+$19, DEK19], which only perform a *logical shift*, as opposed to the *arithmetic shift* that is needed to ensure correctness on signed inputs.

Handling garbled circuits. Our conversion method can also be extended to convert binary shares to garbled circuits, putting the 'Y' into 'ABY' and allowing constant round binary computations. In this paper, we do not focus on this, since the technique is exactly the same as described in [AOR$^+$19]; when using binary shares based on TinyOT MACs, conversions between binary and garbled circuit representation comes for free, based on the observation from Hazay et al. [HSS17] that TinyOT sharings can be locally converted into shares of a multi-party garbled circuit.

Performance evaluation.

We have implemented our protocol in all relevant security models and computation domains as provided by MP-SPDZ [Kel20], and we found it reduces communication both in microbenchmarks and application benchmarks when comparing to a purely arithmetic or a daBit-based implementation. More concretely, for secure comparisons the reduction in communication lies between a factor of 2 and 60 going from purely arithmetic to edaBits, and between 2 and 25 from daBits to edaBits. Improvements in throughput per second are slightly lower but still as high as a factor of 47. Generally, the improvements are higher for dishonest-majority computation and semi-honest security when considering black-box approaches such as purely arithmetic computation or using daBits. However, semi-honest computation allows for non-black-box approaches [MR18, DSZ15] that are as fast as ours.

We have also compared our implementation with the most established software for mixed circuits [BDK$^+$18] and found that it still improves up to a factor of two for a basic benchmark in semi-honest two-party computation. However, they maintain an advantage if the parties are far apart (100 ms RTT) due to the usage of garbled circuits.

Finally, a comparison with a purely arithmetic implementation of deep-learning inference shows an improvement of up to a factor six in terms of both communication and wall clock time.

3.2 Related Work

A (classic) daBit is defined as a pair $([b]_M, [b]_2)$, where $b \in \{0,1\}$ is a random bit. We make use of these daBits to convert one single bit from the binary world to the arithmetic world. Classic daBits can be preprocessed as described in [AOR$^+$19, RW19, RST$^+$19, DEF$^+$19], for example. First, we review at a very high level how these methods work. Then, in Section 3.2, we present the explicit protocols we use in our implementation for generating daBits, and their relation to the works we mentioned above.

Marbled Circuits [RW19].

Each party proposes a set of daBits, whose consistency is checked via cut-and-choose techniques. Then these bits are XORed together to output the final daBits. This method works for both $M = p$ and $M = 2^k$ with minor modifications.

Zaphod [AOR$^+$19].

First arithmetic shares of random bits are produced. Then these are converted to binary shares by observing that the overflow bits in the arithmetic world are rather predictable if the shares are only between two parties. The resulting binary-shared bits may not be correct, so a consistency check is put in place. This works by taking a linear random combination of the bits in both worlds and checking its consistency (in the arithmetic world the LSB must be extracted, which requires an extra sub-protocol). This method is suited for $M = p$.

Actively Secure Setup for SPDZ [RST$^+$19].

This work considers a much more general concept of daBits in which bits can be shared modulo many different primes. The layout of the protocol is similar to the one from Zaphod: Random bits are generated modulo a large-enough prime, and these are converted locally to shares over the integers. Then these are converted to shares modulo each desired prime, and their correctness is checked via linear combinations. Since in [RST$^+$19] the odd primes may be small, the authors have to consider a variant of the subset sum problem to argue security. When instantiating their method with 2 and our large prime p, we notice that their methods essentially lead to an optimized version of Zaphod (in fact, when the odd primes are large enough one can avoid the subset-sum assumption entirely by masking the upper bits as done in Zaphod).

SPDZ2k [DEF$^+$19].

The tools presented in this work are enough to produce daBits, although the authors do not consider this concept explicitly. In a nutshell, this approach would follow the exact same template as in Zaphod, making use of the fact that in SPDZ2k, the parties can obtain binary additive shares of an arithmetically-shared bit b by simply considering the LSB of their shares. Compare this to the field case, where the overflow bit mod p must be predicted and corrected. Furthermore, one can also observe that in SPDZ2k opening the LSB of an arithmetically shared value is also efficient and does not require any overhead with respect to opening the full value (in fact, it is more efficient), unlike the field case.

Our daBit Implementation

Our daBit generation over a prime p is similar to the one considered in Zaphod [AOR$^+$19]. However, we modify the first step in which arithmetic shares of a random bit are produced. Instead of using the random-bit generation from SPDZ, we let each party share an arithmetic bit and then these will be added to produce the desired bit. This is presented in Fig. 3.1. The result is trivially correct if all parties are honest. Furthermore, as the number of participants is larger

Generation of faulty daBits

Pre:

- $\mathcal{F}_{\mathsf{ABB}}$
- Threshold t (maximal number of corrupted parties)

Post: supposed daBit $([b]_M, [b]_2)$

1. $t+1$ parties (w.l.o.g $P_1, \ldots, P_{t+1}$) each input a bit b_i into $\mathcal{F}_{\mathsf{ABB}}$ both mod M and mod 2, resulting in $([b_i]_M, [b_i]_2)$ for $i = 1, \ldots, t+1$.
2. All parties compute $([b]_M, [b]_2) = ([\bigoplus_{i=1}^{t+1} b_i]_M, [\bigoplus_{i=1}^{t+1} b_i]_2)$. The first half can be computed using the fact that $a \oplus b = a + b - 2ab$ for $a, b \in \{0,1\} \subset \mathbb{Z}$ while the second is straight-forward given that $a \oplus b = a + b$ for $a, b \in \mathbb{Z}_2$.

Figure 3.1: Protocol to generate supposed daBits in any domain

than the number of corrupted parties, the result is a random bit from the view of the adversary in that case. The protocol costs t multiplications in $\mathcal{F}_{\mathsf{ABB}}$.

We also notice that if the arithmetic modulus is a power of two, it is easy to construct a daBit from a random bit by having the parties input the least significant bit of their share to the binary computation and then computing the XOR without communication. In other words, parties can locally convert an additive secret sharing modulo 2^k locally. Let b_i denote an additive share of b modulo 2^k held by P_i. Then, $b_i \bmod 2$ is a valid share of b modulo 2:

$$\sum (b_i \bmod 2) \bmod 2 = \left(\sum b_i \bmod 2^k\right) \bmod 2 = b \bmod 2.$$

This is precisely how Zaphod converts from modulo p to modulo 2, but they do not consider the modulo 2^k case. We present this optimization in Fig. 3.2. Furthermore, as a bonus, we observe that in the honest majority setting where no MAC are required this procedure can be made much simpler, and we present this in Fig. 3.3

Note that our protocol for SPDZ2k is more general than the one proposed by Damgård et al. [DEF+19] because theirs only works if the binary part of $\mathcal{F}_{\mathsf{ABB}}$ is implemented by SPDZ2k for $k = 1$, which has the disadvantage that computing an AND has cost quadratic in the security parameter s whereas the protocol by Frederiksen et al. [FKOS15] for example has linear cost in that regard while achieving the same security properties.

In our construction two things must be checked to prevent cheating from an active adversary. First, as in Zaphod, parties may cause the final daBit to be inconsistent, in the sense that the arithmetic and binary parts may contain different bits. Second, unlike the construction from Zaphod, it is not guaranteed that the value each party inputs is indeed a bit.

To fix the first issue we simply resort to the same technique as in Zaphod of computing s random linear combinations modulo two in both domains, after which s daBits have to be discarded for privacy. This method has asymptotically no overhead in terms of daBits being produced because the batch can be arbitrarily large. On the other hand, to fix the second issue, we check that the arithmetic part of each of the final daBits contains indeed a bit, which can be done by checking $x(1-x) = 0$ with x being the arithmetic share. This adds one multiplication per daBit. Furthermore, we notice that we are checking that the final daBit contains a bit, rather

SPDZ2k daBit generation

Pre:

1. $\mathcal{F}_{\mathsf{ABB}}$ with the arithmetic part based on SPDZ2k
2. Total number of parties n

Post: supposed daBit $([b]_{2^k}, [b]_2)$ where $[b]_{2^k}$ is guaranteed to be in $\{0, 1\}$

1. The parties generate a random bit $[b]_{2^k}$ as described by Damgård et al. [DEF+19].
2. Let b_i denote the additive share of b held by P_i, that is $b = \sum_{i=1}^{n} b_i \mod 2^k$. P_i inputs b_i mod 2 to the binary part of $\mathcal{F}_{\mathsf{ABB}}$.
3. The parties compute $[b]_2 = \bigoplus_{i=1}^{n} [b_i \bmod 2]_2$.

Figure 3.2: Protocol to generate supposed daBits with SPDZ2k

daBit generation modulo in $\mathbb{Z}_{2^k}$ without MAC

Pre: $\mathcal{F}_{\mathsf{ABB}}$ where the arithmetic part is based on purely on additive or replicated secret sharing and the binary part uses the same secret sharing scheme

Post: supposed daBit $([b]_{2^k}, [b]_2)$ where $[b]_{2^k}$ is guaranteed to be in $\{0, 1\}$

1. The parties generate a random bit $[b]_{2^k}$ in the arithmetic part of $\mathcal{F}_{\mathsf{ABB}}$.
2. Let $\{b_i^1, \dots, b_i^m\}$ denote the shares of b held by P_i. P_i computes $\{b_i^1 \bmod 2, \dots, b_i^m \bmod 2\}$ and uses them as shares for the binary part of $\mathcal{F}_{\mathsf{ABB}}$.

Figure 3.3: Protocol to generate supposed daBits in protocols module 2^k without MAC

than checking that each of the original daBits input by each party contain a bit. This is more efficient and it is also secure, as there is at least one honest party who inputs a bit, and therefore the XOR operation becomes an oblivious selection between x or $1 - x$, where x is the XOR of the arithmetic shares of the adversary. If the result is a bit, then x was a bit to begin with.

Fig. 3.4 shows our adapted checking protocol. Aly et al. argue that any incorrect daBit would lead to a 1/2 probability of failure in step 1c, hence s independent repetitions would fail at least once with overwhelming probability. They also argue that discarding s daBits after the checks protects the secrecy of the remaining ones.

Paper Outline

We begin in Section 3.3 with some preliminaries. In Section 3.4, we introduce edaBits and show how to instantiate them, given a source of private edaBits. We then present our protocol for creating private edaBits in Section 3.5, based on the new cut-and-choose procedure. Then, in Sections 3.5–3.5 we describe abstract cut-and-choose games that model the protocol, and carry out a formal analysis. Then in Section 3.6 we show how to use edaBits for higher-level primitives like comparison and truncation. Finally, in Section 3.8, we analyze the efficiency of our constructions and present performance numbers from our implementation.

daBit check

Pre: m supposed bits $([b_i]_M, [b_i]_2)$ in $\mathcal{F}_{\mathsf{ABB}}$ where $m > s$ for statistical security parameter s

Post: $m - s$ verified daBits

1. The parties do the following s times:
   a) Generate m fresh public random bits r_i
   b) Compute $[\bigoplus_{i=1}^{m} r_i \cdot b_i]_2$ and open it.
   c) Compute $[r] := [\sum_{i=1}^{m} r_i \cdot b_i]_M$.
      - If $M = 2^k$, call $r' = \mathsf{open}([r \cdot 2^{k-1}]_{2^k})$ and compute $r'/2^{k-1} = (r \cdot 2^{k-1} \bmod 2^k)/(2^{k-1}) = r \bmod 2$.
      - If $M = p$, call $r' = \mathsf{open}([r]_p + 2 \cdot \sum_{i=0}^{s+1} [c_i]_p \cdot 2^i)$ with random bits $[c_i]_p$ and compute $r \bmod 2 = r' \bmod 2$.

   Abort if $r \bmod 2$ does not match the bit from the previous step.
2. Discard $([b_i]_M, [b_i]_2)$ for $i \in [m - s + 1, m]$.
3. For $i \in [1, m - s]$, compute and open $[b_i \cdot (1 - b_i)]_M$. Abort if any value is not zero.[a]

[a]This check may be omitted if $M = 2^k$ and the bit generation via SPDZ2k from Fig. 3.2 is used.

Figure 3.4: Protocol to check classic daBits

3.3 Preliminaries

In this work we consider three main algebraic structures: $\mathbb{Z}_M$ for $M = p$ where p is a large prime, $M = 2^k$ where k is a large integer, and $\mathbb{Z}_2$.

Arithmetic Black-Box

We model MPC via the arithmetic black box model (ABB), which is an ideal functionality in the universal composability framework [Can01]. This functionality allows a set of n parties $P_1, \ldots, P_n$ to input values, operate on them, and receive outputs after the operations have been performed. Typically (see for example Rotaru and Wood [RW19]), this functionality is parameterized by a positive integer M, and the values that can be processed by the functionality are in $\mathbb{Z}_M$, with the native operations being addition and multiplication modulo M.

In this work, we build on the basic ABB to construct edaBits, which are used in our higher-level applications. We therefore consider an extended version of the arithmetic black box model that handles values in both binary and arithmetic domains. First, within one single instance of the functionality we can have both binary and arithmetic computations, where the latter can be either modulo p or modulo 2^k. Furthermore, the functionality allows the parties to convert a single binary share into an arithmetic share of the same bit (but not the other way round). We will use this limited conversion capability to bootstrap to our fully-fledged edaBits, which can convert larger ring elements in both directions, and with much greater efficiency. The details of the functionality are presented in Fig. 3.5.

Functionality $\mathcal{F}_{\mathsf{ABB}}$

Input: On input $(\mathsf{Input}, P_i, \mathsf{type}, \mathsf{id}, x)$ from P_i and $(\mathsf{Input}, P_i, \mathsf{type}, \mathsf{id})$ from all other parties, with id a fresh identifier, $\mathsf{type} \in \{\mathsf{binary}, \mathsf{arithmetic}\}$ and $x \in \mathbb{Z}_2$ or $x \in \mathbb{Z}_M$ (depending on type), store $(\mathsf{type}, \mathsf{id}, x)$.

Linear Combination: On input $(\mathsf{LinComb}, \mathsf{type}, \mathsf{id}, (\mathsf{id}_j)_{j=1}^m, \mathsf{type}, c, (c_j)_{j=1}^m)$, where each id_j is stored in memory and $c, c_j \in \mathbb{Z}_2$ if $\mathsf{type} = \mathsf{binary}$ or $c, c_j \in \mathbb{Z}_M$ if $\mathsf{type} = \mathsf{arithmetic}$, retrieve $((\mathsf{type}, \mathsf{id}_1, x_1), \dots, (\mathsf{type}, \mathsf{id}_m, x_m))$, compute $y = c + \sum_j x_j \cdot c_j$ modulo 2 if $\mathsf{type} = \mathsf{binary}$ and modulo M if $\mathsf{type} = \mathsf{arithmetic}$, and store $(\mathsf{type}, \mathsf{id}, y)$.

Multiply: On input $(\mathsf{Mult}, \mathsf{type}, \mathsf{id}, \mathsf{id}_1, \mathsf{id}_2)$ from all parties (where $\mathsf{id}_1, \mathsf{id}_2$ are present in memory), retrieve $(\mathsf{type}, \mathsf{id}_1, x)$, $(\mathsf{type}, \mathsf{id}_2, y)$, compute $z = x \cdot y$ modulo 2 if $\mathsf{type} = \mathsf{binary}$ and modulo M if $\mathsf{type} = \mathsf{arithmetic}$, and store (id, z).

From Binary to Arithmetic: On input $(\mathsf{ConvertB2A}, \mathsf{id}, \mathsf{id}')$ from all parties, retrieve $(\mathsf{binary}, \mathsf{id}', x)$ and store $(\mathsf{arithmetic}, \mathsf{id}, x)$.

Output: On input $(\mathsf{Output}, \mathsf{type}, \mathsf{id})$ from all honest parties (where id is present in memory), retrieve $(\mathsf{type}, \mathsf{id}, y)$ and output it to the adversary. Wait for an input from the adversary; if this is $\mathsf{Deliver}$ then output y to all parties, otherwise output Abort.

Figure 3.5: Ideal functionality for the MPC arithmetic black box modulo 2 and modulo M, where M is either 2^k or p.

Notation.

As shorthand, we write $[x]_2$ to refer to a secret bit x that has been stored by the functionality $\mathcal{F}_{\mathsf{ABB}}$, and similarly $[x]_M$ for a value $x \in \mathbb{Z}_M$ with $M \in \{p, 2^k\}$. We overload the operators $+$ and $\cdot$, writing for instance, $[y]_M = [x]_M \cdot [y]_M + c$ to denote that the secret values x and y are first multiplied using the Mult command, and then the public constant c is added using $\mathsf{LinComb}$.

Instantiations.

There are several ways to instantiate the basic commands of this functionality, depending on the adversarial setting. In the honest majority setting one can use for example Shamir secret-sharing or replicated-secret sharing [DN07, BLW08], which can be either passively or actively secure [FLNW17]. In the dishonest majority setting, additive secret-sharing is typically used. For the case of active security, we can combine this with information-theoretic MACs to enforce correct opening of shared values [DPSZ12, DKL$^+$13, CDE$^+$18, WRK17b]. Furthermore, the conversions between the arithmetic bits and binary sharings can be implemented via daBits, as shown in [AOR$^+$19, RW19, RST$^+$19]. We present the protocol for daBit generation in Fig. 3.1.

Since all of these are linear secret-sharing schemes, when secret values inside $\mathcal{F}_{\mathsf{ABB}}$ represent sharings under such a scheme, the $\mathsf{LinComb}$ command of $\mathcal{F}_{\mathsf{ABB}}$ can be implemented by simply computing the same linear combination on the shares. The Mult command is usually realized by preprocessing multiplication triples, that is, shared values $[a]_M, [b]_M, [c]_M$ where a, b are uniformly random in $\mathbb{Z}_M$ and $c = a \cdot b$. Given such a triple, two secret values $[x]_M, [y]_M$ can be multiplied by first opening $x + a$ and $y + b$, and then computing

Functionality $\mathcal{F}_{\mathsf{edaBits}}$

The functionality is parametrized by $M \in \{2^k, p\}$ and $m \leq \log M$. It has the same features as $\mathcal{F}_{\mathsf{ABB}}$, together with the following command:

Create edaBits: On input $(\mathsf{edabit}, \mathsf{id}_M, \mathsf{id}_2)$ from all parties, sample $(r_0, \dots, r_{m-1}) \in \mathbb{Z}_2^m$ uniformly at random and store $(\mathsf{binary}, \mathsf{id}_2, r_j)$ for $j = 0, \dots, m-1$, together with $(\mathsf{arithmetic}, \mathsf{id}_M, r)$, where $r = \sum_{j=0}^{m-1} r_j 2^j$.

Figure 3.6: Ideal functionality for extended daBits.

$$[z]_M = (x+a)(y+b) - (x+a)[b]_M - (y+b)[a]_M + [c]_M$$

which can be computed as a linear operation in the secret values, producing $z = xy$.

We remark that preprocessing triples is often the most expensive part of the entire MPC protocol, especially in the dishonest majority setting. In the arithmetic case, these can be produced using linearly or somewhat homomorphic encryption [DPSZ12, KPR18], oblivious linear function evaluation [DGN+17] or, with a higher communication cost, oblivious transfer [KOS16, CDE+18]. In the binary case with $M = 2$, techniques based on oblivious transfer are usually fastest, and these are known as the TinyOT family of protocols [NNOB12, FKOS15, WRK17a, WRK17b].

3.4 Extended daBits

The main primitive of our work is the concept of extended daBits, or *edaBits*. Unlike a daBit, which is a random bit b shared as $([b]_M, [b]_2)$, an edaBit is a collection of bits $(r_{m-1}, \dots, r_0)$ such that (1) each bit is secret-shared as $[r_i]_2$ and (2) the integer $r = \sum_{i=0}^{m} r_i 2^i$ is secret-shared as $[r]_M$.

One edaBit of length m can be generated from m daBits, and in fact, this is typically the first step when applying daBits to several non-linear primitives like truncation. Instead of following this approach, we choose to generate the edaBits—which is what is needed for most applications where daBits are used—directly, which leads to a much more efficient method and ultimately leads to more efficient primitives for MPC protocols.

At a high level, our protocol for generating edaBits proceeds as follows. Let us think initially of the passively secure setting. Each party P_i samples m random bits $r_{i,0}^i, \dots, r_{i,m-1}^i$, and secret-shares these bits towards the parties over $\mathbb{Z}_2$, as well as the integer $r_i = \sum_{j=0}^{m-1} r_{i,j} 2^j$ over $\mathbb{Z}_M$. Since each edaBit is known by one party, these edaBits must be combined to get edaBits where no party knows the underlying values. We refer to the former as *private* edaBits, and to the latter as *global* edaBits. The parties combine the private edaBits by adding them together: the arithmetic shares can be simply added locally as $[r]_M = \sum_{i=1}^{n} [r_i]_M$, and the binary shares can be added via an n-input binary adder. Some complications arise, coming from the fact that the r_i values may overflow mod p. Dealing with this is highly non-trivial, and we will discuss this in detail in the description of our protocol in Section 3.4. However, before we dive into our construction, we will first present the functionality we aim at instantiating. This functionality is presented in Fig. 3.6.

Functionality for Private Extended daBits

We also use a functionality $\mathcal{F}_{\mathsf{edaBitsPriv}}$, which models a *private* set of edaBits that is known to one party. This functionality is defined exactly as $\mathcal{F}_{\mathsf{edaBits}}$, except that the bits $r_0, \dots, r_{m-1}$ are given as output to one party; additionally, if that party is corrupt, the adversary may instead choose these bits.

The heaviest part of our contribution lies on the instantiation of this functionality, which we postpone to Section 3.5.

From Private to Global Extended daBits

As we discussed already at the beginning of this section, one can instantiate $\mathcal{F}_{\mathsf{edaBits}}$ using $\mathcal{F}_{\mathsf{edaBitsPriv}}$, by combining the different private edaBits to ensure no individual party knows the underlying values. Small variations are required depending on whether $M = 2^k$ or $M = p$, for reasons that will become clear in a moment.

Now, to provide an intuition on our protocol, assume that the ABB is storing $([r_i]_M, [r_{i,0}]_2, \dots, [r_{i,m-1}]_2)$ for $i = 1, \dots, n$, where party P_i knows $(r_{i,0}, \dots, r_{i,m-1})$ and $r_i = \sum_{j=1}^{m-1} r_{i,j} 2^j$. The parties can add their arithmetic shares to get shares of $r' = \sum_{i=1}^{n} r_i \mod M$, and they can also add their binary shares using a binary n-input adder, which results in shares of the bits of r', only without modular reduction.

Since we want to output a random m-bit integer, the parties need to remove the bits of r' beyond the m-th bit from the arithmetic shares. We have binary shares of these carry bits as part of the output from the binary adder, so using $\log(n)$ calls to ConvertB2A of $\mathcal{F}_{\mathsf{ABB}}$, each of which costs a (regular) daBit, we can convert these to the arithmetic world and perform the correction. Notice that for the case of $M = 2^k$, $m = k$, we can omit this conversion since the arithmetic shares are already reduced.

Even without the correction above, the least significant m bits of r' still correspond to $r_0, \dots, r_{m-1}$. This turns out to be enough for some applications because it is easy to "delete" the most significant bit in $\mathbb{Z}_{2^k}$ by multiplying with two. We call such an edaBit loose as apposed to a strict one as defined in Fig. 3.6.

One must be careful with potential overflows modulo M. If $M = 2^k$, then any overflow bits beyond the k-th position can simply be discarded. On the other hand, if $M = p$, as long as $m < \log p$ then we can still subtract the $\log n$ converted carries from the arithmetic shares to correct for any overflow modulo p. The protocol is given in Fig. 3.7, and the security stated in Theorem 3.1 below, whose proof follows in a straightforward manner from the correctness of the additions in the protocol. In the protocol, nBitADD denotes an n-input binary adder on m-bit inputs. This can be implemented naively in a circuit with $< (m + \log n) \cdot (n - 1)$ AND gates.

Theorem 3.1. *Protocol Π_{edaBits} UC-realizes functionality $\mathcal{F}_{\mathsf{edaBits}}$ in the $(\mathcal{F}_{\mathsf{edaBitsPriv}}, \mathcal{F}_{\mathsf{B2A}})$-hybrid model.*

3.5 Instantiating Private Extended daBits

Our protocol for producing private edaBits is fairly intuitive. The protocol begins with each party inputting a set of edaBits to the ABB functionality. However, since a corrupt party may input inconsistent edaBits (that is, the binary part may not correspond to the bit representation of the arithmetic part), some extra checks must be set in place to ensure correctness. To this end, the parties engage in a consistency check, where each party must prove that their private

Protocol Π_{edaBits}

Pre:

- Access to $\mathcal{F}_{\mathsf{edaBitsPriv}}$.
- If $M = p$, then $0 < m < \log(p)$.

Post: The parties get $([r]_M, [r_i]_2, \ldots, [r_i]_2)$ where $r = \sum_{j=1}^{m-1} r_i 2^j$ and the bits are uniform to the adversary.

1. The parties call the functionality $\mathcal{F}_{\mathsf{edaBitsPriv}}$ to get random shares $([r_i]_M, [r_{i,0}]_2, \ldots, [r_{i,m-1}]_2)$, for $i = 1, \ldots, n$. Party P_i additionally learns $r_{i,j}$ and $r_i = \sum_{j=1}^{m-1} r_{i,j} 2^j$.
2. The parties invoke $\mathcal{F}_{\mathsf{ABB}}$ to compute $[r']_M = \sum_{i=1}^{n} [r_i]_M$.
3. The parties invoke $\mathcal{F}_{\mathsf{ABB}}$ to compute $\mathsf{nBitADD}\left(([r_{1,j}]_2)_j, \ldots, ([r_{n,j}]_2)_j\right)$, obtaining $m + \log n$ bits $([b_0]_2, \ldots, [b_{m+\log(n)-1}]_2)$.
4. Call $\mathsf{ConvertB2A}$ from $\mathcal{F}_{\mathsf{ABB}}$ to convert $[b_j]_2 \mapsto [b_j]_M$ for $j = m, \ldots, m + \log(n) - 1$. If $M = 2^k$, values b_j for $j > k$ do not need to be converted, and for the sake of notation, we denote $[b_j]_{2^k} := 0$ for $j > k$.
5. Use $\mathcal{F}_{\mathsf{ABB}}$ to compute $[r]_M = [r']_M - 2^m \sum_{j=0}^{\log(n)-1} [b_{j+m}]_M 2^j$.
6. Output $([r]_M, [b_0]_2, \ldots, [b_{m-1}]_2)$.

Figure 3.7: Protocol for generating global edaBits from private edaBits.

edaBits were created correctly. We do this with a cut-and-choose procedure, where first a random subset of a certain size of edaBits is opened, their correctness is checked, and then the remaining edaBits are randomly placed into buckets. Within each bucket, all edaBits but the first one are checked against the first edaBit by adding the two in both the binary and arithmetic domains, and opening the result. With high probability, the first edaBit will be correct if all the checks pass.

This method is based on a standard cut-and-choose technique for verifying multiplication triples, used in several other works [FKOS15, FLNW17]. However, the main difference in our case is that the checking procedure for verifying two edaBits within a bucket is much more expensive: checking two multiplication triples consists of a simple linear combination and openeing, whereas to check edaBits, we need to run a binary addition circuit on secret-shared values. This binary addition itself requires $O(m)$ multiplication triples to verify, and the protocol for producing these triples typically requires further cut-and-choose steps to ensure correctness and security.

In this work, we take a different approach to reduce this overhead. First, we allow some of the triples used to perform the check within each bucket to be incorrect, which saves in resources as a triple verification step can be omitted. Furthermore, we observe that these multiplication triples are intended to be used on inputs that are known to the party proposing the edaBits, and thus it is acceptable if this party knows the bits of the underlying triples as well. As a result, we can simplify the triple generation by letting this party sample the triples together with the edaBits, which is much cheaper than letting the parties jointly sample (even incorrect) triples. Note that

Protocol $\Pi_{\mathsf{edaBitsPriv}}$

Pre: $\mathcal{F}_{\mathsf{ABB}}$ with modulus M, length parameter $m \in \mathbb{Z}$ with $m \leq \log_2 M$
Post: Batch of N shared edaBits $\{([r_j]_M, [r_{j,0}]_2, \dots, [r_{j,m-1}]_2)\}_{j=1}^N$, where party P_i knows the underlying bits.

1. P_i samples $r_{j,0}, \dots, r_{j,m-1} \in \mathbb{Z}_2$, for $j = 1, \dots, NB + C$, and inputs these to $\mathcal{F}_{\mathsf{ABB}}$ in $\mathbb{Z}_2$.
2. P_i computes $r_j = \sum_{i=0}^{m-1} r_{j,i} 2^i$ and inputs $r_j \in \mathbb{Z}_M$ to $\mathcal{F}_{\mathsf{ABB}}$.
3. P_i samples $(N(B-1) + C')m$ random bit triples and inputs these to $\mathcal{F}_{\mathsf{ABB}}$.
4. The parties run the CutNChoose procedure to check the consistency of these edaBits. If the check passes, then the parties obtain N edaBits. Otherwise, they abort.

Figure 3.8: Protocol for producing private extended daBits.

even though the triples may be incorrect, they must still be authenticated (in practice, with MACs) by the party who proposes them so that the errors cannot be changed after generating the triples.

To model this, we extend the arithmetic black box model with the following commands, for generating a private triple, and for faulty multiplication, which uses a previously stored triple to do a multiplication.

Input Triple. On input $(\mathsf{Triple}, \mathsf{id}, a, b, c)$ from P_i, where id is a fresh binary identifier and $a, b, c \in \{0, 1\}$, store $(\mathsf{Triple}, i, \mathsf{id}, a, b, c)$.

Faulty Multiplication. On input $(\mathsf{FaultyMult}, \mathsf{id}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_T, i)$ from all parties (where $\mathsf{id}_1, \mathsf{id}_2$ are present in memory), retrieve $(\mathsf{binary}, \mathsf{id}_1, x)$, $(\mathsf{binary}, \mathsf{id}_2, y)$, $(\mathsf{Triple}, i, \mathsf{id}_T, a, b, c)$, compute $z = x \cdot y \oplus (c \oplus a \cdot b)$, and store (id, z).

The triple command can be directly instantiated using Input from $\mathcal{F}_{\mathsf{ABB}}$, while FaultyMult uses Beaver's multiplication technique with one of these triples. Note that in Beaver-based binary multiplication, it is easy to see that any additive error in a triple leads to exactly the same error in the product.

Now we are ready to present our protocol to preprocess private edaBits, described in Fig. 3.8. The party P_i locally samples a batch of edaBits and multiplication triples, then inputs these into $\mathcal{F}_{\mathsf{ABB}}$. The parties then run the CutNChoose subprotocol, given in Fig. 3.9, to check that the edaBits provided by P_i are consistent. The protocol outputs a batch of N edaBits, and is parametrized by a bucket size B, and values C, C' which determine how many edaBits and triples are opened, respectively. BitADDCarry denotes a two-input binary addition circuit with a carry bit, which must satisfy the weakly additively tamper resilient property given in the next section. As we will see later, this can be computed with m AND gates and depth $m - 1$.[1]

The cut-and-choose protocol starts by using a standard coin-tossing functionality, $\mathcal{F}_{\mathsf{Rand}}$, to sample public random permutations used to shuffle the sets of edaBits and triples. The coin-tossing

[1]This circuit is rather naive, and in fact there are logarithmic depth circuits with a greater number of AND gates. However, as we will see later in the section, it is important for our security proof to use specifically these naive circuits to obtain the tamper-resilient property. Furthermore, they are only used in the preprocessing phase, so the overhead in round complexity is insignificant in practice.

Procedure CutNChoose

Pre: A batch of $(NB + C)$ shared edaBits of the form $\{([r]_M, [r_0]_2, \dots, [r_{m-1}]_2)\}$ and a batch of $(N \cdot (B-1) \cdot m + C' \cdot m)$ triples, all stored in $\mathcal{F}_{\mathsf{ABB}}$, where party P_i knows the underlying bits of the edaBits and the triples.
Post: N verified edaBits
The parties do the following:

1. Using $\mathcal{F}_{\mathsf{Rand}}$, sample two public random permutations and use these to shuffle the edaBits and the triples.
2. Open the first C of the shuffled edaBits in both worlds, and the first $C' \cdot m$ triples. Abort if any of the edaBits or the triples are inconsistent.
3. Place the remaining edaBits into buckets of size B and the triples into buckets of size $(B-1) \cdot m$.
4. For each bucket, select the first edaBit $([r]_M, [r_0]_2, \dots, [r_{m-1}]_2)$, and for every other edaBit $([s]_M, [s_0]_2, \dots, [s_{m-1}]_2)$ in the same bucket, perform the following check:
   a) Let $[r+s]_M = [r]_M + [s]_M$.
   b) Let $([c_0]_2, \dots, [c_m]_2) = \mathsf{BitADDCarry}([r_0]_2, \dots, [r_{m-1}]_2, [s_0]_2, \dots, [s_{m-1}]_2)$, using the FaultyMult command to evaluate each AND gate.
   c) Convert $[c_m]_2 \mapsto [c_m]_M$ with ConvertB2A.
   d) Let $[c']_M = [r+s]_M - 2^m \cdot [c_m]_M$. Open c' and the corresponding bits $c_0, \dots, c_{m-1}$ from the binary world, and check that $c' = \sum_{i=0}^{m-1} c_i 2^i$.
5. If all the checks pass, output the first edaBit from each of the N buckets.

Figure 3.9: Cut-and-choose procedure to check correctness of input edaBits.

can be implemented, for example, with hash-based commitments in the random oracle model. Then the first C edaBits and $C'm$ triples are opened and tested for correctness; this is to ensure that not too large a fraction of the remaining edaBits and triples are incorrect. Then the edaBits are divided into buckets of size B, together with $B-1$ sets of m triples in each bucket. Then, the top edaBit from each bucket is checked with every other edaBit in the bucket by evaluating a binary addition circuit using the triples, and comparing the result with the same addition done in the arithmetic domain. Each individual check in the CutNChoose procedure takes two edaBits of m bits each, and consumes m triples as well as a single regular daBit, needed to convert the carry bit from the addition into the arithmetic domain. Note that when working with modulus $M = 2^k$, if $m = k$ then this conversion step is not needed.

Weakly Tamper-Resilient Binary Addition Circuit

To implement the BitADDCarry circuit we use a ripple-carry adder, which computes the carry bit at every position with the following equation:

$$c_{i+1} = c_i \oplus ((x_i \oplus c_i) \wedge (y_i \oplus c_i)), \forall i \in \{0, m-1\} \quad (3.1)$$

where $c_0 = 0$, and x_i, y_i are the i-th bits of the two binary inputs. It then outputs $z_i = x_i \oplus y_i \oplus c_i$, for $i = 0, \dots, m-1$, and the last carry bit c_m. Note that this requires m AND gates and has linear depth.

Below we define the tamper-resilient property of the circuit that we require. We consider an adversary who can additively tamper with a binary circuit by inducing bit-flips in the output wires of any AND gate.

Definition 3.1. *A binary circuit $\mathcal{C} : \mathbb{F}_2^{2m} \to \mathbb{F}_2^{m+1}$ is* weakly additively tamper resilient*, if given any tampered circuit $\mathcal{C}^*$, obtained by additively tampering $\mathcal{C}$, one of the following holds:*

1. $\forall (x, y) \in \mathbb{F}_2^m : \ \mathcal{C}(x, y) = \mathcal{C}^*(x, y)$.
2. $\forall (x, y) \in \mathbb{F}_2^m : \ \mathcal{C}(x, y) \neq \mathcal{C}^*(x, y)$.

Intuitively, this says that the tampered circuit is either incorrect on every possible input, or functionally equivalent to the original circuit. In our protocol, this property restricts the adversary from being able to pass the check with a tampered circuit with bad edaBits as well as the same circuit with good edaBits. It ensures that if any multiplication triple is incorrect, then the check at that position would only pass with either a good edaBit, or a bad edaBit (but not both).

We now show that this property is satisfied by the ripple-carry adder circuit above, which we use.

Lemma 3.1. *The ripple carry adder circuit above is weakly additively tamper-resilient (Definition 3.1).*

Proof. Consider a tampered circuit $\mathcal{C}^*$, and let i be the smallest index where the AND gate in equation 3.1 has been tampered. Since c_i was computed correctly, we have $\mathcal{C}^*(x, y)[i + 1] = \mathcal{C}(x, y)[i + 1] \oplus 1$. Therefore, any tampering leads to incorrect output, so the circuit is weakly additively tamper resilient. □

As a side note, the naive binary circuit which requires 2 AND gates per carry computation also has the property of being weakly additively tamper resilient. Because it has 2 AND gates, it can either be the case that $\mathcal{C}(x, y) = \mathcal{C}^*(x, y)$ or $\mathcal{C}(x, y) = \mathcal{C}^*(x, y) \oplus 1$, depending on whether the carry computation was tampered with 1 triple or 2 triples. In either case, this is independent of x and y.

In the case of generating edaBits over $\mathbb{Z}_p$, we still use the ripple-carry adder circuit, and our protocol works as long as the length of the edaBits satisfies $m < \log(p)$. If we wanted edaBits with $m = \lceil \log p \rceil$, for instance to be able to represent arbitrary elements of the field, it seems we would need to use an addition circuit modulo p. Unfortunately, the natural circuit consisting of a binary addition followed by a conditional subtraction is *not* weakly additively tamper resilient. One possible workaround is to use Algebraic Manipulation Detection (AMD) [GIP+14, GIW16] circuits, which satisfy much stronger requirements than being weakly additively tamper resilient, however this gives a very large overhead in practice.

Overview of Cut-and-Choose Analysis

The remainder of this section is devoted to proving that the cut-and-choose method used in our protocol is sound, as stated in the following theorem.

Table 3.1: Number of edaBits produced by CutNChoose for statistical security 2^{-s} and bucket size B, with $C = C' = B$.

s	B	# of edaBits
40	3	≥ 1048576
40	4	≥ 10322
40	5	≥ 1024
80	5	≥ 1048576

Theorem 3.2. *Let $N \geq 2^{s/(B-1)}$ and $C = C' = B$, for some bucket size $B \in \{3, 4, 5\}$. Then the probability that the* CutNChoose *procedure in protocol $\Pi_{\mathsf{edaBitsPriv}}$ outputs at least one incorrect* edaBit *is no more than 2^{-s}.*

Assuming the theorem above, we can prove that our protocol instantiates the desired functionality, as stated in the following theorem. The only interesting aspect to note about security is that we need $m \leq \log M$ to ensure that the value c' computed in step 4d of CutNChoose does not overflow modulo p when $M = p$ is prime. This guarantees that the check values are computed the same way in the binary and arithmetic domains.

Theorem 3.3. *Protocol $\Pi_{\mathsf{edaBitsPriv}}$ securely instantiates the functionality $\mathcal{F}_{\mathsf{edaBitsPriv}}$ in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model.*

To give some idea of parameters, in Table 3.1 we give the required bucket sizes and number N of edaBits that must be produced to ensure 2^{-s} failure probability according to Theorem 3.2. Note that these are exactly the same bounds as the standard cut-and-choose procedure without any faulty verification steps from [FLNW17]. Our current proof relies on case-by-case analyses for each bucket size, which is why Theorem 3.2 is not fully general. We leave it as an open problem to obtain a general result for any bucket size.

Overview of Analysis.

We analyse the protocol by looking at two abstract games, which model the cut-and-choose procedure. The first game, RealGame, models the protocol fairly closely, but is difficult to directly analyze. We then make some simplifying assumptions about the game to get SimpleGame, and show that any adversary who wins in the real protocol can be translated into an adversary in the SimpleGame. This is the final game we actually analyze.

Abstracting the Cut-and-Choose Game

We first look more closely at the cut-and-choose procedure by defining an abstract game, RealGame, shown in Figure 3.10, that models this process. Note that in this game, the only difference compared with the original protocol is that the adversary directly chooses additively tampered binary circuits, instead of multiplication triples. The check procedure is carried out exactly as before, so it is clear that this faithfully models the original protocol.

Complexities of analyzing the game. In this game, the adversary can pass the check with a bad edaBit in two different ways. The first is to corrupt edaBits in multiples of the bucket size B, and hope that they all end up in the same bucket so that the errors cancel each other out. The

RealGame

1. $\mathcal{A}$ prepares $NB+C$ edaBits $\{(r_j, r_{j,0}, \ldots, r_{j,m-1})\}_{j=1}^{NB+C}$, and batch of $N(B-1)+C'$ potentially tampered circuits $\{\mathcal{C}^*{}_j\}_{j=1}^{N(B-1)+C'}$ to send to the challenger.
2. The challenger shuffles the edaBits and the circuits using 2 permutations.
3. The challenger opens C edaBits in both worlds and C' circuits randomly. If any of the edaBits are inconsistent, or the circuits have been tampered, Abort.
4. Within each bucket, for every pair of edaBits $(r, (r_i)_i)$ and $(s, (s_i)_i)$, take the next circuit C^* and compute $(c_0, \ldots, c_m) = \mathcal{C}^*(r_0, \ldots, r_{m-1}, s_0, \ldots, s_{m-1})$. Compute $c = \sum_{i=0}^{m-1} c_i 2^i$ and check that $r + s - 2^m \cdot c_m$ equals c.

The adversary wins if all the checks pass and there is at least one corrupted edaBit in the output.

Figure 3.10: Abstract game modelling the actual cut-and-choose procedure

second way is to corrupt a set of edaBits and guess the permutation in which they are most likely to end up. Once a permutation is guessed, the adversary will know how many triples it needs to corrupt in order to cancel out the errors, and must also hope that the triples end up in the right place.

To compute the exact probability of all these events, we will also have to consider the number of ways in which the bad edaBits can be corrupted. For edaBits which are m bits, there are up to $2^m - 1$ different ways in which they may be corrupted. On top of that, we have to consider the number of different ways in which these bad edaBits may be paired in the check. In order to avoid enumerating the cases and the complex calculation involved, we simplify the game in a few ways which can only give the adversary a better chance of winning. However, we show that these simplifications are sufficient for our purpose.

The SimpleGame

In this section we analyze a simplified game and bound the success probability of any adversary in that game by 2^{-s}. Before explaining the simple game, we will leave the complicated world of edaBits and triples. We define a TRIP to be a set of triples that is used to check two edaBits. In our simple world edaBits transform into balls, *GOOD* edaBits into white balls (○) and *BAD* edaBits into gray balls (●). An edaBit is *BAD* when at least one of the underlying bits are not correct. TRIPs transform themselves into triangles, *GOOD* TRIPs into white triangles (△) and *BAD* TRIPs into gray triangles (▲). We define a TRIP to be *BAD* when it helps the adversary to win the game, in other words if it can alter the result of addition of two edaBits. Figure 3.11 illustrates the simple game.

In the SimpleGame $\mathcal{A}$ wins if there is no Abort (means $\mathcal{A}$ passes all the checks) and there is at least one bad ball in the final output. The **simple** BucketCheck checks all the buckets. Precisely, in each bucket two balls are being checked using one triangle. For example, let us consider the size of the buckets $B = 3$. Now one bucket contains three balls $[B1, B2, B3]$ and two triangles $[T1, T2]$. Then BucketCheck checks if the configurations $[B1, B2|T1]$ and $[B1, B3|T2]$ matches any one of these configurations $\{[○, ○|▲], [○, ●|△], [●, ○|△]\}$. If that is the case then BucketCheck

SimpleGame

1. $\mathcal{A}$ prepares $NB + C$ balls, corrupts b of them and sends them to the challenger.
2. The challenger opens C of them randomly and checks whether all of them are good. If any one of them is not good, Abort.
3. The challenger permutes and throws NB balls into N buckets each of size B uniformly at random. Then sends the order of arrangement to $\mathcal{A}$.
4. $\mathcal{A}$ prepares $N(B-1) + C'$ triangles, corrupts t of them and sends them to the challenger.
5. The challenger opens C' of them randomly and checks whether all of them are good. If any one of them is not good, Abort.
6. The challenger permutes and throws $N(B-1)$ triangles into N buckets uniformly at random and runs the **Simple** BucketCheck subroutine.
7. If **Simple** BucketCheck returns 1, the challenger outputs first ball from each bucket. Else, Abort.

$\mathcal{A}$ wins if there is no Abort and at least one bad ball is in the output.

Figure 3.11: Simplified CutNChoose game

Aborts. When there are two bad balls and one triangle the abort condition depends on the type of bad balls. That means we are considering all bad balls to be distinct, say with different color shades. As a result, in some cases challenger aborts if the checking configuration matches [●, ●|▲] and in other cases it aborts due to [●, ●|△] configuration.

In the simple world everyone has access to a public function f, which takes two bad balls and a triangle as input and outputs 0 or 1. If the output is zero, that means it is a bad configuration, otherwise it is good. This function is isomorphic to the check from step 4 of RealGame, which takes 2 edaBits and a circuit as inputs and outputs the result of the check. The BucketCheck procedure uses f to check all the buckets. Figure 3.12 illustrates the check in detail. $\mathcal{A}$ passes BucketCheck if all the check configurations are favorable to the adversary. These favorable check configurations are illustrated in Table 3.2.

After throwing triangles, in each bucket, if the check configuration of balls and triangles are from the first three entries of Table 3.2, then BucketCheck will not Abort. For the last entry BucketCheck will not Abort if the output of f is 1. Notice that if BucketCheck passes only due to the first configuration of Table 3.2 in all buckets, then the output from each bucket is going to be

Table 3.2: Favorable combination of balls and triangles for the adversary.

Balls		Triangles
○	○	△
○	●	▲
●	○	▲
●	●	△/▲

Simple BucketCheck

Input: N buckets and a function f. Each bucket contains B balls $\{x_1, \ldots, x_B\}$ and $(B-1)$ triangles $\{y_1, \ldots, y_{B-1}\}$.
Output: 0 or 1.
Runs this check in each bucket:

1. Check the configuration of $[x_1, x_i|y_{i-1}]$ $\forall i \in [2, B]$.
   - If $[x_1, x_i|y_{i-1}] \in \{[\circ, \circ|\blacktriangle], [\circ, \bullet|\triangle], [\bullet, \circ|\triangle]\}$ return Reject.
   - If $[x_1, x_i|y_{i-1}] \in [\bullet, \bullet|\triangle]$ and $f(\bullet, \bullet, \triangle) = 0$ return Reject.
   - If $[x_1, x_i|y_{i-1}] \in [\bullet, \bullet|\blacktriangle]$ and $f(\bullet, \bullet, \blacktriangle) = 0$ return Reject.
2. Otherwise return Accept.

If check returns Accept for all the buckets, then output 1; Otherwise output 0.

Figure 3.12: A simple bucket check procedure

a good ball and $\mathcal{A}$ loses. So ideally we should take that into account while computing the winning probability of the adversary. However, for most of the cases it is sufficient to show that for large enough N the Pr[$\mathcal{A}$ passes BucketCheck] is negligible in the statistical security parameter s, as that will bound the winning probability of $\mathcal{A}$ in the simple game.

Before analyzing the SimpleGame, we show that security of RealGame follows directly from security of SimpleGame. Intuitively, that is indeed the case, as in the SimpleGame an adversary chooses number of bad triangles adaptively; Whereas in the RealGame it has to fix the tampered circuits before seeing the permuted edaBits. Thus, if an adversary cannot win the SimpleGame then it must be more difficult for it to succeed in the RealGame.

Lemma 3.2. *Security against all adversaries in* SimpleGame *implies security against all adversaries in* RealGame*.*

Proof. (Sketch.) We prove that by showing if there exist an efficient adversary $\mathcal{B}$ that wins RealGame with non-negligible probability, then there exist an efficient adversary $\mathcal{A}$ against the SimpleGame challenger that wins the game with non-negligible probability. $\mathcal{A}$ simulates the challenger of the RealGame and uses $\mathcal{B}$ to win the SimpleGame. $\mathcal{B}$ sends a batch of edaBits and a set of circuits to $\mathcal{A}$. $\mathcal{A}$ transforms the edaBits into circles. It randomly permutes the circuits, and transforms them into triangles. Clearly, a ball (or triangle) is good or bad depends on whether that was a good or bad edaBit (or a circuit).

$\mathcal{A}$ sends the set of balls to the SimpleGame challenger. The challenger throws them randomly in buckets, sends the arrangement to $\mathcal{A}$. Then $\mathcal{A}$ sends the set of triangles to the challenger. The challenger throws them randomly in buckets, and sends the arrangement to $\mathcal{A}$. In the RealGame $\mathcal{A}$ throws edaBits and the circuits according to the arrangement of balls and triangles in the SimpleGame. Clearly, the simulation is indistinguishable from a RealGame challenger. Thus from the final distribution of triangles, $\mathcal{B}$ cannot distinguish whether it is in the RealGame or in the simulation. Also in the SimpleGame the BucketCheck uses the public function f, which is isomorphic to check function that takes as input two edaBits and a circuit and outputs the result of the check, from step 4 of the RealGame. Consequently, if $\mathcal{B}$ wins with non-negligible probability then $\mathcal{A}$ wins the SimpleGame with a non-negligible probability. □

Throughout the analysis, we use b to denote the number of bad balls and t to denote the number of bad triangles. Now in order to win the SimpleGame the adversary has to pass all the three checks, so let us try to bound the success probability of $\mathcal{A}$ for each of them. Throughout the analysis we consider $N \geq 2^{\frac{s}{B-1}}$, that is for $B \geq 3$, $N(B-1) \geq 2^{\frac{s}{B-1}+1}$ and we are opening $B(\geq 3)$ balls and B triangles in the first two checks.

Opening C balls:

In the first check the challenger opens C balls and check whether they are good. So,

$$\Pr[C \text{ balls are good}] = \frac{\binom{NB+C-b}{C}}{\binom{NB+C}{C}} \approx (1 - b/(NB+C))^C.$$

Now for $b = (NB+C)\alpha$, where $1/(NB+C) \leq \alpha \leq 1$, the probability can be written as $(1-\alpha)^C$. In order to bound the success probability of the adversary with the statistical security parameter s, let us consider the case when $\alpha \geq \frac{2^{s/B}-1}{2^{s/B}}$ and $C = B$. Thus,

$$\Pr[C \text{ balls are good}] \approx (1-\alpha)^C = (2^{-s/B})^B = 2^{-s}.$$

So if the challenger opens B balls to check then in order to pass the first check $\mathcal{A}$ must corrupt less than α fraction of the balls, where $\alpha = \frac{2^{s/B}-1}{2^{s/B}}$. Lemma 3.3 follows from the above analysis.

Lemma 3.3. *The probability of $\mathcal{A}$ passing the first check in* SimpleGame *is less than 2^{-s}, if the adversary corrupts more than α fraction of balls for $\alpha = \frac{2^{s/B}-1}{2^{s/B}}$ and the challenger opens B balls.*

Opening C' triangles:

In this case we'll consider the probability of $\mathcal{A}$ passing the second check. This is similar to the previous check, the only difference is that here the challenger opens C' triangles and checks whether they are good. Consequently,

$$\Pr[C' \text{ triangles are good}] = \frac{\binom{N(B-1)+C'-t}{C'}}{\binom{N(B-1)+C'}{C'}} \approx (1 - t/(N(B-1)+C'))^{C'}.$$

As in the previous case, if t is more than β fraction of the total number of triangles for $\beta = \frac{2^{s/B}-1}{2^{s/B}}$, we can upper bound the success probability of $\mathcal{A}$ by $(2^{-s/B})^{C'}$. Thus for $C' = B$ the success probability of $\mathcal{A}$ in the second check can be bounded by 2^{-s}. Lemma 3.4 follows from the above analysis.

Lemma 3.4. *The probability of $\mathcal{A}$ passing the second check in* SimpleGame *is less than 2^{-s}, if the adversary corrupts more than β fraction of triangles for $\beta = \frac{2^{s/B}-1}{2^{s/B}}$ and the challenger opens B triangles.*

Lemmas 3.3–3.4 show that it suffices to only look at the first two checks to prove security when the fraction of bad balls or bad triangles is sufficiently large. However, when one of these is small, we also need to analyze the checks within each bucket in the game.

BucketCheck procedure:

In this case we consider that the adversary passes first two checks and reaches the last level of the game. However, in order to win the game the adversary has to pass the BucketCheck. Note that now we are dealing with NB balls and the challenger already fixes the arrangement of NB balls in N buckets. Once the ball permutation is fixed that imposes a restriction on the number of favorable (for $\mathcal{A}$) triangle permutations. For example, let us consider that the challenger throws 12 balls into 4 buckets of size 3 and fixes this permutation:

$$\{[●, ○, ○][○, ○, ●][●, ●, ○][○, ○, ○]\}$$

Then there are only two possible favorable permutations of triangles:

$$\{[▲, ▲][△, ▲][△, ▲][△, △]\}$$
$$\{[▲, ▲][△, ▲][▲, ▲][△, △]\}$$

Two favorable permutations come from the fact that the third bucket contains two bad balls. From Table 3.2 we can see that whenever there are two bad balls in a bucket the adversary can pass the check in that bucket either with a good triangle or with a bad triangle. That means both configurations $[●, ●|△]$ and $[●, ●|▲]$ might be favorable to the adversary. Now $\mathcal{A}$ can use the public function f to determine the value of $f(●, ●, △)$ and $f(●, ●, ▲)$. In this example, let us consider the value of $f(●, ●, △)$ to be 1; Then the first permutation of triangles is favorable to the adversary. As a result the probability of passing the BucketCheck essentially depends on the probability of hitting that specific permutation of triangles among all possible arrangements of triangles. Then the probability of the adversary passing the last check given a specific arrangement of balls L_i is given by:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}|L_i] = 1/\binom{N(B-1)}{t}$$

where $t = N(B-1)\beta$. Thus,

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}|L_i] = \frac{(N(B-1)\beta)!\,(N(B-1)(1-\beta))!}{N(B-1)!}$$

In order to upper bound $\Pr[\mathcal{A}$ passes BucketCheck$]$ we will upper bound the probability for different ranges of α and β. Note that the total probability is given by:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] = \sum_i \Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}|L_i] \cdot \Pr[L_i]$$

If we can argue that for all possible $(2^{s/B}-1)/2^{s/B} \geq \alpha \geq 1/NB$, the maximum probability for $\Pr[\mathcal{A}$ passes BucketCheck$|L_i]$, for some configuration L_i, can be bounded by 2^{-s}, then:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \sum_i 2^{-s} \cdot \Pr[L_i]$$

Note that the maximum possible value of α is 1, however as the challenger opens C balls and C' triangles, the adversary cannot set α to be 1. To pass the first check $\mathcal{A}$ must set α to be less than $(2^{s/B}-1)/2^{s/B}$ if the challenger opens B balls and B triangles.

Now let us try to bound $\Pr[\mathcal{A}$ passes BucketCheck$|L_i]$. The value of $\binom{N(B-1)}{t}$ maximizes at $t \approx N(B-1)/2$. Starting from the case when there is no bad triangle, the probability monotonically decreases from 1 to its minimum at $\beta \approx 1/2$, and then it monotonically increases to 1 when all triangles are bad. We analyze the success probability of $\mathcal{A}$ in three cases.

Case I ($B-1 \le t \le N(B-1)-(B-1)$)**:** Here we are considering the cases when $\mathcal{A}$ chooses number of bad triangles t from the range $[B-1, N(B-1)-(B-1)]$ to maximize its success probability. Now,

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}|L_i] = 1/\binom{N(B-1)}{t}.$$

Clearly, the probability is maximum when t is equal to $(B-1)$ or $N(B-1)-(B-1)$, which is given by:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}|L_i] = \frac{(B-1)! \cdot (N(B-1)-(B-1))!}{N(B-1)!}$$
$$= \left(\frac{B-1}{N(B-1)}\right) \cdot \left(\frac{B-2}{N(B-1)-1}\right) \cdots \left(\frac{1}{N(B-1)-(B-2)}\right)$$

Now given $N \ge 2^{\frac{s}{B-1}}$ we have,

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}|L_i] \le \left(\frac{1}{2^{\frac{s}{B-1}}}\right)^{B-1} = 2^{-s}.$$

Thus for a given b if the adversary chooses number of bad triangles $t \in [B-1, N(B-1)-(B-1)]$, then:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \le \sum_i 2^{-s} \cdot \Pr[L_i].$$

Given b bad balls and $(NB-b)$ good balls one can arrange them in $NB!/(NB-b)!$ ways. So the probability of hitting a specific arrangement L_i is $(NB-b)!/NB!$. Thus:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \le \frac{NB!}{(NB-b)!} \cdot 2^{-s} \cdot \frac{(NB-b)!}{NB!} = 2^{-s}.$$

Case II ($t > (N(B-1)-(B-1))$)**:** If t is greater than $(N(B-1)-(B-1))$ then the adversary will not be able to pass the second check as the challenger opens $C' = B$ triangles. Thus,

$$\Pr[C' \text{ triangles are good}] = \frac{\binom{N(B-1)+C'-t}{C'}}{\binom{N(B-1)+C'}{C'}} \le \left(1 - \frac{t}{N(B-1)+B}\right)^B$$
$$\le \left(\frac{2B-2}{N(B-1)+B}\right)^B = \left(\frac{2}{N}\right)^B \cdot \left(\frac{B-1}{B-1+\frac{B}{N}}\right)^B,$$

which is less than 2^{-s} given $N \ge 2^{\frac{s}{B-1}}$ and $\frac{s}{B-1} > B$.

Case III ($t < B-1$)**:** Here we try to find the best strategy for the adversary and then show that the success probability can be bounded by 2^{-s} if $N \ge 2^{s/B-1}$. We analyze the probability for three sub-cases, specifically for bucket size 3, 4 and 5, as that allows us to use our cut and choose technique for a wide range of practical parameters.

Bucket size 3: For bucket size 3 we have to consider two cases, namely $t = 0$ and $t = 1$. Let us first consider the case when $t = 0$. Clearly, if $\mathcal{A}$ corrupt all the $NB + C$ balls in a way such that f(●, ●, △) always returns 1, then the adversary trivially passes BucketCheck. However in that case $\mathcal{A}$ fails with probability 1 as the challenger opens B balls in the first check. If $\mathcal{A}$ corrupts α fraction of NB balls, where $\alpha \geq \frac{2^{s/B}-1}{2^{s/B}}$; Then the success probability of the $\mathcal{A}$ can be bounded by 2^{-s}, if the challenger opens $C = B$ balls in the first check, given $N \geq 2^{\frac{s}{B-1}}$. To pass the first check $\mathcal{A}$ can only corrupt less than α fraction of NB balls. However, in that case the total number of good balls are more than one. Notice that if there is even one good ball out of the NB balls, then in the BucketCheck [●, ○|△] or [○, ●|△] check configuration occurs for most of L_is, and $\mathcal{A}$ fails. More precisely, whenever the number of bad balls are not multiple of B, then there exist a bucket with a good ball and a bad ball, thus probability of $\mathcal{A}$ passing BucketCheck becomes zero. When number of bad balls are multiple of B then there exist very few configurations for which the probability of $\mathcal{A}$ passing the BucketCheck is one; For all other possible combinations it become zero. As an example, for $(B = 3, N = 3, K = 2, t = 0)$ only one type of configuration is favorable for the adversary when K is fixed, where K is the number of bad balls to be outputted at the end of the BucketCheck, thus $1 \leq K \leq N - 1$:

{[●, ●, ●][●, ●, ●][○, ○, ○]}

Since f(●, ●, △) returns 1, we can assume that all the bad balls are corrupted in the same way. Let us consider $b = KB$, then:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \frac{\binom{N}{K}}{\binom{NB}{KB}}$$

At $K \approx N/2$ this probability reaches its minimum value $2^{-(NB-1)} \ll 2^{-s}$. At $K = 1$ and $K = (N - 1)$ the probability reaches its maximum value which is less than $(B - 1)!/(NB - (B - 1))^{B-1} \leq 2^{-s}$ for $B \geq 3$ as $NB > 2^{s/2}$. Given that, the best strategy of the adversary would be to corrupt one bucket, so that it can pass the first check and hope to hit a favorable configuration in the BucketCheck. However, in that case the probability is still negligible in s. Note that the analysis for this case is same as the one from [FLNW17].

For $t = 1$ the analysis is very much similar to the previous case. Only difference is that now the adversary has to compensate for that one bad triangle. In this case the adversary can win only when the number of bad balls b are KB, $KB - 1$ or $KB + 1$ for $1 \leq K \leq (N - 1)$. We are not considering the case when K is N, as in that $\mathcal{A}$ passing the first check is $\mathsf{negl}(s)$. For example for $(B = 3, N = 4, K = 2, t = 1)$, these are three possible type of favorable configurations for the adversary:

{[●, ●, ●][●, ⚫, ●][○, ○, ○][○, ○, ○]}
{[●, ●, ●][●, ●, ○][○, ○, ○][○, ○, ○]}
{[●, ●, ●][●, ●, ●][○, ●, ○][○, ○, ○]}

In the first case there must exist exactly one bad ball pair in one corrupted bucket such that f(●, ⚫, ▲) returns 1, thus for that pair the adversary can use the bad triangle. In the second case the adversary uses the bad triangle to check one {bad ball, good ball} pair in the second bucket. In a similar way in the third case $\mathcal{A}$ uses the bad triangle to check one {good ball, bad ball} pair in the third bucket. Note that in the second case the good ball in the second bucket can be placed in four possible positions to generate other favorable permutations. Similarly in the third case the bad ball in the third bucket can be placed in four possible positions to generate other favorable permutations. For all other arrangement the adversary fails BucketCheck, as it has to deal with more than one {bad ball, good ball} pair.

Now the probability of $\mathcal{A}$ passing the BucketCheck for the case when $b = KB$ and $t = 1$ is given by:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \frac{\binom{N}{K}}{\binom{NB}{KB}} \cdot (B-1) \cdot K \cdot \frac{1}{N(B-1)}.$$

The probability of $\mathcal{A}$ passing the BucketCheck when $b = KB - 1$ and $t = 1$ is given by:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \frac{\binom{N}{K}}{\binom{NB}{KB-1}} \cdot (B-1) \cdot K \cdot \frac{1}{N(B-1)}.$$

In the last case for $b = KB + 1$ and $t = 1$ the probability is given by:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \frac{\binom{N}{K}}{\binom{NB}{KB+1}} \cdot (B-1) \cdot (N-K) \cdot \frac{1}{N(B-1)}.$$

The probability of success in the second case for $K = 1$ is higher than the probabilities in the first case for all possible K. In fact it maximizes at $K = 1$ in the second case; which is the same as the highest probability in the third case when $K = (N-1)$. Consequently the best strategy of the adversary would be to corrupt minimum number of balls, to minimize the failure probability of opening and checking C balls, and try to achieve the maximum success probability from the BucketCheck. That means the optimal strategy for the adversary would be the second case with $K = 1$. Thus,

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \frac{(B-1)!}{(NB-(B-2))^{B-1}} \leq 2^{-s}, \ \textit{for } B \geq 3.$$

Bucket size 4**:** The analysis for the cases $B = 4$, $t = 0$ and $t = 1$ follows directly from the analysis from $B = 3$. In other words, the configurations remain the same, the only difference being the bucket size is now 4.

For bucket size $B = 4$ and $t = 2$, there are six possible favorable configurations for the adversary when K is fixed, where K is the number of bad balls to be outputted at the end of the BucketCheck, thus $1 \leq K \leq N-1$. For example for $(B = 4, N = 4, K = 2, t = 2)$ these are the six possible configurations for the adversary:

{[●, ●, ●, ●][●, ●, ●, ○][○, ○, ●, ○][○, ○, ○, ○]}
{[●, ●, ●, ●][●, ●, ○, ○][○, ○, ○, ○][○, ○, ○, ○]}
{[●, ●, ●, ●][●, ●, ●, ●][○, ○, ●, ○][○, ●, ○, ○]}
{[●, ●, ●, ●][●, ●, ●, ⚫][○, ○, ●, ○][○, ○, ○, ○]}
{[●, ●, ●, ●][●, ●, ⚫, ○][○, ○, ○, ○][○, ○, ○, ○]}
{[●, ●, ●, ●][●, ●, ⚫, ⚫][○, ○, ○, ○][○, ○, ○, ○]}

For all these cases the success probability of the adversary in the BucketCheck can be expressed as:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \binom{N}{K}\binom{K(B-1)}{g_1+b_1}\binom{(N-K)(B-1)}{b_2}\frac{1}{\binom{NB}{KB-g_1+b_2}}\frac{1}{\binom{N(B-1)}{t}}, \tag{3.2}$$

where g_1 is the total number of good balls in the chosen K buckets (which output bad balls at the end of BucketCheck), b_1 is the total number of different kind of bad balls(●), such that f(●, ●, ▲) returns 1 and b_2 is the total number of bad balls from other $(N-K)$ buckets. Note that number of bad triangles t is equal to $g_1 + b_1 + b_2$. As an example let us consider the first configuration among the six favorable configurations; In that case $g_1 = 1$, $b_1 = 0$ and $b_2 = 1$, thus:

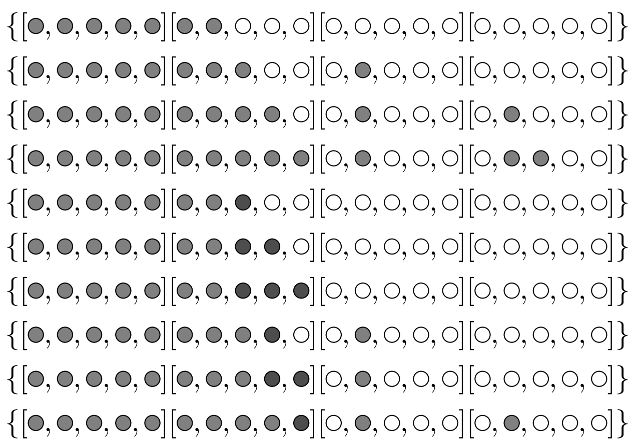
$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \binom{N}{K}\binom{K(B-1)}{1}\binom{(N-K)(B-1)}{1}\frac{1}{\binom{NB}{KB}}\frac{1}{\binom{N(B-1)}{t}}.$$

Now for each of these configurations the probability is maximum either at $K = 1$ or at $K = N-1$. After calculating the probabilities for each of these configurations at $K = 1$ and $K = N-1$ it is easy to see that the success probability of the adversary is maximum in the fourth case for $K = N-1$. Thus:

$$\begin{aligned}
&\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq 9N(N-1)\cdot\frac{1}{\binom{4N}{3}}\cdot\frac{1}{\binom{3N}{2}}\\
&= \left(\frac{3N-3}{3N-1}\right)\cdot\left(\frac{3}{4N}\right)\cdot\left(\frac{2}{4N-1}\right)\cdot\left(\frac{2}{4N-2}\right)\\
&\leq \left(\frac{3N-3}{3N-1}\right)\cdot 2^{-s/3}\cdot 2^{-s/3}\cdot 2^{-s/3} \leq 2^{-s},\ \textit{given}\ N \geq 2^{s/B-1}.
\end{aligned}$$

Bucket size 5: Once again, the analysis from the previous cases carries over for $t = 0, 1, 2$, $t = 3$ being the only new case we have to analyze.

For the case when B = 5 and t = 3, there are 10 favorable configurations for the adversary when K is fixed. For N = 4 and K = 2, these are the cases:



Using eq. (3.2) we can calculate the probabilities for these 10 cases at $K = 1$ and $K = N-1$ to find the best case scenario for the adversary. Doing so, we found that the first case from the figure at $K = 1$, and the fourth case at $K = N-1$ have the best probabilities. Considering the first case, this would be,

$$
\begin{aligned}
&\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \binom{N}{1} \cdot \binom{4}{3} \cdot \frac{1}{\binom{5N}{2}} \cdot \frac{1}{\binom{4N}{3}} \\
&= N \cdot 4 \cdot \frac{1}{\binom{5N}{2}} \cdot \frac{1}{\binom{4N}{3}} \\
&= \left(\frac{2}{5N}\right) \cdot \left(\frac{6}{5N-1}\right) \cdot \left(\frac{1}{4N-1}\right) \cdot \left(\frac{1}{4N-2}\right) \\
&\leq 2^{-s/4} \cdot 2^{-s/4} \cdot 2^{-s/4} \cdot 2^{-s/4} \leq 2^{-s}, \ \textit{given } N \geq 2^{s/B-1}.
\end{aligned}
$$

Even though the probability is the same for the fourth case with $K = N - 1$, since the number of bad balls are much higher than the first case, the overall probability will be lower for the fourth, making the first case the best one.

We summarize the analysis as follows.

Lemma 3.5. *The probability of $\mathcal{A}$ passing the* BucketCheck *in* SimpleGame *is less than 2^{-s}, if $N \geq 2^{s/(B-1)}$ and the challenger opens $C = B$ balls and $C' = B$ triangles during first two checks of* SimpleGame *for $B \in \{3, 4, 5\}$ given $\frac{s}{B-1} > B$.*

Proof. This follows from the case-by-case analysis of BucketCheck procedure, together with Lemma 3.3 and Lemma 3.4. □

Combining Lemma 3.2 and Lemma 3.5, this completes the proof of Theorem 3.2.

Remark 3.1. *As we already mentioned the bound we obtain is not general. However, from Lemma 3.5 it is evident that one can produce more than* 1024 edaBits *efficiently with* 40*-bit statistical security using different bucket sizes with our* CutNChoose *technique, which is sufficient for the applications we are considering in this work. It also shows that if we want to achieve* 80*-bit statistical security for $N \geq 2^{20}$, then increasing the bucket size from* 3 *to* 5 *would be sufficient. Table 3.1 shows the number of* edaBits *we can produce with different size of buckets.*

Optimizing Parameters

In this part of the analysis we try to find optimal parameters for the number of edaBits (C) and the number of TRIPs (C') we need to open so that probability of the adversary winning the game is at least $1/2^s$, where s is the security parameter.

Recall that the probability of the adversary winning the game has four components:

$$\Pr[\text{Winning}] = \Pr[C] \cdot \Pr[C'] \cdot \Pr[\mathsf{TRIPs}] \cdot \Pr[\mathsf{edaBits}]$$

Calculating the exact probability Pr[edaBits] is hard because it requires calculating the number ways in which red balls can be permuted for a given number of red balls. For a small number of balls, this is doable but as the number of balls increase beyond a certain threshold, this becomes challenging.

On the other hand, if we can prove that the probability of the other three components together is *negligible*, we do not need to compute the exact probability for the red balls. The fact that it is < 1 is enough for us to work with.

Expanding the other three components of the probability:

$$\Pr[C] \cdot \Pr[C'] \cdot \Pr[\mathsf{TRIPs}] =$$

$$= \frac{(p+C-b)!}{(p-b)!} \cdot \frac{p!}{(p+C)!} \cdot \frac{(q+C'-t)!}{(q-t)!} \cdot \frac{q!}{(q+C')!} \cdot \frac{t! \cdot (q-t)!}{q!} \cdot 2^l$$

$$= \frac{(p+C-b)!}{(p-b)!} \cdot \frac{p!}{(p+C)!} \cdot \frac{(q+C'-t)!}{(q+C')!} \cdot t! \cdot 2^l$$

We split the calculation into two cases based on how big k and k' are.

Recalling Stirling's Approximation: $n! \sim \sqrt{2\pi n} \cdot (n/e)^n$. For $n = 100$, the ratio of the approximation to the value of $n!$ is 0.999167. For our calculations this is accurate enough.

For the sake of clarity, expand the numerator and the denominator individually. Applying the approximation to the numerator first:

$$\begin{aligned} = \sqrt{2\pi(p+C-b)} \cdot \left[\frac{p+C-b}{e}\right]^{(p+C-b)} \cdot \sqrt{2\pi p} \cdot \left[\frac{p}{e}\right]^{p} \\ \cdot \sqrt{2\pi(q+C'-t)} \cdot \left[\frac{q+C'-t}{e}\right]^{(q+C'-t)} \cdot t! \cdot 2^l \end{aligned} \tag{3.3}$$

The denominator becomes:

$$\begin{aligned} \sqrt{2\pi(p-b)} \cdot \left[\frac{p-b}{e}\right]^{(p-b)} \cdot \sqrt{2\pi(p+C)} \cdot \left[\frac{p+C}{e}\right]^{(p+C)} \sqrt{2\pi(q+C')} \cdot \\ \left[\frac{q+C'}{e}\right]^{(q+C')} \end{aligned} \tag{3.4}$$

There are same number of $\sqrt{2\pi}$ terms the numerator and the denominator. Canceling those and isolating the terms we get,

$$t! \cdot 2^l \cdot \sqrt{\frac{p+C-b}{p-b} \cdot \frac{p}{p+C} \cdot \frac{q+C'-t}{q+C'}} \tag{3.5}$$

$$\frac{e^{(p-b)+(p+C)+(q+C')}}{e^{(p+c-b)+p+(q+C'-t)}} \tag{3.6}$$

$$\frac{(p+C-b)^{(p+C-b)}}{(p-b)^{(p-b)}} \cdot \frac{p^p}{(p+C)^{(p+C)}} \cdot \frac{(q+C'-t)^{(q+C'-t)}}{(q+C')^{(q+C')}} \tag{3.7}$$

In equation 3.5, the terms $p/(p+C)$ and $(q+C'-t)/(q+C')$ are always less than 1 because $k \geq 1$. But since they are under the square root and minuscule compared to other terms, they won't impact the parameters significantly and hence are ignored. $(p+C-b)/(p-b)$ is > 1 and also small, but we retain it for now. Computing the powers of e and writing 3.5 and 3.6 together as,

$$t! \cdot 2^l \cdot \sqrt{\frac{p+C-b}{p-b}} \cdot e^t \tag{3.8}$$

In equation 3.7, we can group the terms that have the same power together and write it as,

$$\left[\frac{p+C-b}{p+C}\cdot\frac{p}{p-b}\right]^{p}\cdot\left[\frac{q+C'-t}{q+C'}\right]^{(q+C')}\cdot\left[\frac{p-b}{p+C-b}\right]^{b}\cdot\left[\frac{p+C-b}{p+C}\right]^{C}\cdot\left[\frac{1}{q+C'-t}\right]^{t} \tag{3.9}$$

In equation 3.9, the every term except the first has a denominator that is bigger than the numerator, which is good for us as it allows us to choose values of C, C' that give us negligible probability. But the first term is not < 1 because $p/(p-b)$ is going to increase as b (or the number of corrupted edaBits increases.

Clearly, this is not enough to compute the parameters or to even show that the probability of the adversary winning the game is negligible. So, at this stage we include the 4th component of the total probability, which the the probability that the edaBits are in the guessed permutation. However, we do not need to compute the exact probability, we only need an upper bound on it.

From here, the analysis is specific to a bucket size of 3. Let's start with the expression for the probability of edaBits being in one particular subset of permutations. We use a, b, c to denote the number of buckets that contain 1, 2, and 3 red balls respectively. Therefore, the total number of red balls in the situation will be $a + 2b + 3c$.

$$\binom{N}{x+y+z}\cdot\frac{(x+y+z)!}{x!\cdot y!\cdot z!}\cdot 2^{y}\cdot\frac{(x+2y+3z)!\cdot(NB-(x+2y+3z))!}{NB!} \tag{3.10}$$

The first term $\binom{N}{x+y+z}$ is the number of ways in which the buckets can be chosen for the permutations. The second term is the number of ways in which we can permute the buckets themselves. For the case of a bucket with 2 red balls, recall that two arrangements [🔴,🔴,🟢] and [🔴,🟢,🔴] are possible, so we get a factor of 2^y. And finally we have the number of ways in which we can permute the red balls amongst themselves and the green balls amongst themselves.

The point at which this value maximizes is challenging to find, and since we only need an upper bound, we use the maximal values for each of the individual terms. Maximizing the term $(x+y+z)!\,/x!\cdot y!\cdot z!$ is straightforward, it maximizes when the denominator is the lowest, which is when $x = y = z$ for any given value of the sum. We let $2^y = 2^w$, where w is the maximum possible value for a given number of red balls. The last term is the same for every permutation of the red balls because it only depends on the number of red balls and green balls rather than the arrangement. We now add this probability into the calculation as well, for the case of $x+y+z = k$. But, for the sake of simplifying the calculations, we use $x + y + z = b$ instead of k. This only increases the adversary's chance of winning.

$$t!\cdot 2^{l}\cdot\sqrt{\frac{p+C-b}{p-b}}\cdot e^{t}\cdot\left[\frac{p+C-b}{p+C}\cdot\frac{p}{p-b}\right]^{p}\cdot\left[\frac{q+C'-t}{q+C'}\right]^{q+C'}\cdot\left[\frac{p-b}{p+C-b}\right]^{b}\cdot\left[\frac{p+C-b}{p+C}\right]^{C}\cdot\left[\frac{1}{q+C'-t}\right]^{t}\binom{N}{b}\cdot\frac{b!}{(b/3!)^{3}}\cdot 2^{b}\cdot\frac{b!\cdot(p-b)!}{p!} \tag{3.11}$$

$$
\begin{aligned}
&= \sqrt{2\pi t}\cdot t^t \cdot 2^{l+b}\cdot \binom{N}{b}\cdot\sqrt{\frac{p+C-b}{p-b}}\cdot\sqrt{\frac{2\pi b}{(2\pi)^3(b/3)^3}}\frac{b^b}{(b/3)^b}\cdot\\
&\left[\frac{p+C-b}{p+C}\frac{p}{p-b}\right]^p\cdot\sqrt{\frac{(2\pi)^2 b(p-b)}{2\pi p}}\cdot\frac{b^b\cdot(p-b)^{p-b}}{p^p}\cdot\\
&\left[\frac{q+C'-t}{q+C'}\right]^{q+C'}\cdot\left[\frac{p-b}{p+C-b}\right]^b\cdot\left[\frac{p+C-b}{p+C}\right]^C\cdot\left[\frac{1}{q+C'-t}\right]^t
\end{aligned}
\tag{3.12}
$$

Canceling and reordering the terms,

$$
\begin{aligned}
&= \binom{N}{b}\cdot\sqrt{\frac{27(p+C-b)t}{bp}}\cdot 2^{l+b}\cdot 3^b\cdot\left[\frac{p+C-b}{p+C}\right]^p\cdot\left[\frac{q+C'-t}{q+C'}\right]^q\cdot\\
&\left[\frac{b}{p+C-b}\right]^b\cdot\left[\frac{t}{q+C'-t}\right]^t\cdot\left[\frac{p+C-b}{p+C}\right]^C\cdot\left[\frac{q+C'-t}{q+C'}\right]^{C'}
\end{aligned}
\tag{3.13}
$$

We can use Stirling's approximation for $\binom{N}{b} = N^b/b!$ and reapplying it on $b!$, we get,

$$
\begin{aligned}
&\frac{1}{b}\cdot\sqrt{\frac{27(p+C-b)t}{2\pi p}}\cdot\frac{N^b}{b^b}\cdot 2^{l+b}\cdot 3^b\cdot\left[\frac{p+C-b}{p+C}\right]^p\cdot\left[\frac{q+C'-t}{q+C'}\right]^q\cdot\\
&\left[\frac{b}{p+C-b}\right]^b\cdot\left[\frac{t}{q+C'-t}\right]^t\cdot\left[\frac{p+C-b}{p+C}\right]^C\cdot\left[\frac{q+C'-t}{q+C'}\right]^{C'}
\end{aligned}
\tag{3.14}
$$

Note that $N^b * 3^b$ can be written as p^b since $NB = p$ and in this case we use buckets of size 3.

$$
\begin{aligned}
&\sqrt{\frac{27(p+C-b)t}{2\pi p}}\cdot 2^{l+b}\cdot 3^b\cdot\frac{1}{b}\cdot\left[\frac{p+C-b}{p+C}\right]^p\cdot\left[\frac{q+C'-t}{q+C'}\right]^q\\
&\left[\frac{p}{p+C-b}\right]^b\cdot\left[\frac{t}{q+C'-t}\right]^t\cdot\left[\frac{p+C-b}{p+C}\right]^C\cdot\left[\frac{q+C'-t}{q+C'}\right]^{C'}
\end{aligned}
\tag{3.15}
$$

Ignoring the first four terms for now, we take a deeper look at the other terms. If we can prove that the product of these 6 terms is negligible across the spectrum of values possible for b, t for some particular values C, C', we can conclude there exists no strategy of corrupting red balls and red triangles when we open a certain number of red balls and red triangles before the game begins.

Through these calculations we want to arrive the minimum possible value for $C + C'$, in the order of a tens. Since p, q are of the order 10^6, at some points we may approximate $p/q \pm C/C'$ to be p/q. An observation about the game is that $b \geq 1$ but t can be 0, a fact we will use in the optimization.

Case 1: Starting with the minimum possible corruption of the balls and triangles, which are (1,2), (2,1) and (3,0). Since the values of $b, t.C, C'$ are small compared to p, q, we can approximate $p + C - b$ to p.

3.6 Primitives

This section describes the high-level protocols we build using our $\mathsf{edaBits}$, both over $\mathbb{Z}_{2^k}$ and $\mathbb{Z}_p$. We focus on secure truncation (Section 3.6) and secure integer comparison (Section 3.6), although our techniques apply to a much wider set of non-linear primitives that require binary circuits for intermediate computations. For example, our techniques also allow us to compute binary-to-arithmetic and arithmetic-to-binary conversions of shared integers, by plugging in our $\mathsf{edaBits}$ into the conversion protocols from [Cd10] and [DEF$^+$19] for the field and ring cases, respectively.

Throughout this section our datatypes are signed integers in the interval $[-2^{\ell-1}, 2^{\ell-1})$. On the other hand, our MPC protocols operate over a modulus $M \geq 2^\ell$ which is either 2^k or a prime p. Given an integer $\alpha \in [-2^{\ell-1}, 2^{\ell-1})$, we can associate to it the corresponding ring element in $\mathbb{Z}_M$ by computing $\alpha \bmod M \in \mathbb{Z}_M$ (modular reduction returns integers in $[0, M)$). We denote this map by $\mathsf{Rep}_M(\alpha)$, and we may drop the sub-index M when it is clear from context. Finally, in the protocols below LT denotes a binary less-than circuit.

Truncation

Recall that our datatypes are signed integers in the interval $[-2^{\ell-1}, 2^{\ell-1})$, represented by integers in $\mathbb{Z}_M$ where $M \geq 2^\ell$ via $\mathsf{Rep}_M(\alpha) = \alpha \bmod M$. The goal of a truncation protocol is to obtain $[y]$ from $[a]$, where $y = \mathsf{Rep}\left(\left\lfloor \frac{\alpha}{2^m} \right\rfloor\right)$ and where $a = \mathsf{Rep}(\alpha)$. This is a crucial operation when dealing with fixed-point arithmetic, and therefore an efficient solution for it has a substantial impact in the efficiency of MPC protocols for a wide range of applications. An important observation is that, as integers, $\left\lfloor \frac{\alpha}{2^m} \right\rfloor = \frac{\alpha - (\alpha \bmod 2^m)}{2^m}$. If M is an odd prime p, this corresponds in $\mathbb{Z}_p$ to $y = (\mathsf{Rep}(\alpha) - \mathsf{Rep}(\alpha \bmod 2^m)) \cdot \mathsf{Rep}(2^m)^{-1}$. Furthermore, $\mathsf{Rep}(\alpha \bmod 2^m) = \alpha \bmod 2^m = a \bmod 2^m$ and $\mathsf{Rep}(2^m) = 2^m$, so $y = \frac{a - (a \bmod 2^m)}{(2^m)^{-1}}$.

Truncation over $\mathbb{Z}_{2^k}$.

Truncation protocols over fields typically exploit the fact that one can divide by powers of 2 modulo p. This is not possible when working modulo 2^k. Instead, we take a different approach. Let $[a]_{2^k}$ be the initial shares, where $a = \mathsf{Rep}(\alpha)$ with $\alpha \in [-2^{\ell-1}, 2^{\ell-1})$ (notice that it may be the case that $\ell < k$). First, we provide a method, $\mathsf{LogShift}$, for computing the *logical* right shift of a by m positions, assuming that $a \in [0, 2^\ell)$. That is, if a is

$$(\underbrace{0, \ldots, 0}_{k-\ell}, \underbrace{a_{\ell-1}, \ldots, a_0}_{\ell}),$$

this procedure will yield shares of

$$(\underbrace{0, \ldots, 0}_{k-\ell+m}, \underbrace{a_{\ell-1}, \ldots, a_m}_{\ell-m}).$$

Then, to compute the arithmetic shift, we use the fact that[2]

$$\left\lfloor \frac{\alpha}{2^m} \right\rfloor \equiv \mathsf{LogShift}_m(a + 2^{\ell-1}) - 2^{\ell-m-1} \bmod 2^k.$$

[2]Notice that we can use the $\mathsf{LogShift}$ method on $a + 2^{\ell-1}$ since, $\alpha + 2^{\ell-1} \in [0, 2^\ell)$, which implies that $(a + 2^{\ell-1}) \bmod 2^k = \alpha + 2^{\ell-1}$ and therefore $(a + 2^{\ell-1}) \bmod 2^k$ is ℓ-bits long, as required.

Logical right shift over $\mathbb{Z}_{2^k}$

Pre:

- $\mathcal{F}_{\mathsf{ABB}}$
- Input $[a]_{2^k}$ where $a \in [0, 2^\ell)$.
- Number of bits to shift m
- edaBit $([r]_{2^k}, [r]_2)$ of length m
- edaBit $([r']_{2^k}, [r']_2)$ of length $\ell - m$

Post: $[y]_{2^k}$, where $y = \mathsf{LogShift}_m(a)$.

1. The parties compute shares of $a \bmod 2^m$ as follows:
   a) Call $c = \mathsf{open}\left(2^{k-m} \cdot ([a]_{2^k} + [r]_{2^k})\right)$
   b) Compute $[v]_2 = \mathsf{LT}((c_i)_{i=k-m+1}^{k}, ([r_i]_2)_{i=0}^{m-1})$
   c) Convert $[v]_2 \mapsto [v]_{2^k}$
   d) Let $[a \bmod 2^m]_{2^k} = 2^m\,[v]_{2^k} - [r]_{2^k} + c/2^{k-m}$.
2. The parties compute the truncation:
   a) Compute $[b]_{2^k} = [a]_{2^k} - ([a]_{2^k} \bmod 2^m)$.
   b) Call $d = \mathsf{open}(2^{k-\ell} \cdot ([b]_{2^k} + 2^m\,[r']_{2^k}))$.
   c) Compute $[u]_2 = \mathsf{LT}((d_i)_{i=k-\ell+m}^{k-1}, ([r'_i]_2)_{i=0}^{\ell-m-1})$
   d) Convert $[u]_2 \mapsto [u]_{2^k}$.[a]
   e) Output $[y]_{2^k} = 2^{\ell-m}\,[u]_{2^k} + d/2^{k-\ell+m} - [r']_{2^k}$

[a]One can optimize this by noticing that we only need shares of u modulo $2^{k-\ell+m}$.

Figure 3.13: Protocol for performing logical right-shift

Now, to compute the logical shift, our protocol begins just like in the field case by computing shares of $a \bmod 2^m$ and subtracting them from a, which produces shares of $(a_{k-1}, \ldots, a_m, 0, \ldots, 0)$. The parties then open a masked version of $a - (a \bmod 2^m)$ which does not reveal the upper $k - \ell$ bits, and then shift to the right by m positions in the clear, and undo the truncated mask. One has to account for the overflow that may occur during this masking, but this can be calculated using a binary LT circuit.

The details of our logical shift protocol are provided in Fig. 3.13, and we analyze its correctness next. First, it is easy to see that $c = 2^{k-m}((a + r) \bmod 2^m)$, so $c/2^{k-m} = (a \bmod 2^m) + r - 2^m v$, where v is set if and only if $c/2^{k-m} < r$. From this we can see that the first part of the protocol $[a \bmod 2^m]_{2^k}$ is correctly computed. Privacy of this first part follows from the fact that $r \bmod 2^m$ completely masks $a \bmod 2^m$ when c is opened.

For the second part, let us write $b = 2^m a'$, then $d = 2^{k-\ell+m}((a' + r') \bmod 2^{\ell-m})$, so $d/2^{k-\ell+m} = a' + r' - 2^{\ell-m}u$, where u is set if and only if $d/2^{k-\ell+m} < r'$, as calculated by the protocol. We get then that $a' = d/2^{k-\ell+m} - r' + 2^{\ell-m}u$, and since a' is precisely $\mathsf{LogShift}_m(a)$,

Probabilistic truncation over $\mathbb{Z}_{2^k}$

Pre:

- $\mathcal{F}_{\mathsf{ABB}}$
- Input $[a]_{2^k}$ where $a \in [0, 2^\ell)$.
- $\ell < k$
- Number of bits to truncate m
- edaBit $([r]_{2^k}, [r]_2)$ of length $(\ell - m)$
- edaBit $([r']_{2^k}, [r']_2)$ of length m
- Random bit $[b]_{2^k}$

Post: $[y]_{2^k}$ where $y = \lfloor a/2^m \rfloor + u$ with $u = 1$ with probability $(a \bmod 2^m)/2^m$.

1. Call $c = \mathsf{open}(2^{k-\ell-1} \cdot ([a]_{2^k} + 2^\ell [b]_{2^k} + 2^m [r]_{2^k} + [r']_{2^k}))$. Write $c = 2^{k-\ell-1} c'$.
2. Compute $[v]_{2^k} = [b \oplus c'_\ell]_{2^k} = [b]_{2^k} + c'_\ell - 2c'_\ell [b]_{2^k}$
3. Output $[y]_{2^k} = (c' \bmod 2^\ell)/2^m - [r]_{2^k} + 2^{\ell-m} [v]_{2^k}$

Figure 3.14: Probabilistic truncation in domain modulo power of two using edaBits

we conclude the correctness analysis.

Probabilistic Truncation. Recall that in the field case one can obtain probabilistic truncation avoiding a binary circuit, which results in a constant number of rounds. Over rings this is a much more challenging task. For example, probabilistic truncation with a constant number of rounds is achieved in ABY3 [MR18], but requires, like in the field case, a 2^s gap between the secret values and the actual modulus, which in turn implies that only small non-negative values can be truncated.

In Fig. 3.14, we take a different approach. Intuitively, we follow the same approach as in ABY3, which consists of masking the value to be truncated with a shared random value for which its corresponding truncation is also known, opening this value, truncating it and removing the truncated mask. In ABY3 a large gap is required to ensure that the overflow that may happen by the masking process does not occur with high probability. Instead, we allow this overflow bit to be non-zero and remove it from the final expression. Doing this naively would require us to compute a LT circuit, but we avoid doing this by using the fact that, because the input is positive, the overflow bit can be obtained from the opened value by making the mask value also positive. This leaks the overflow bit, which is not secure, and to avoid this we mask this single bit with another random bit. This protocol can be seen as an extension of the probabilistic truncation protocol by Dalskov et al. [DEK19]. Below, we provide an analysis for our extension that also applies to said protocol.

Now we analyze the protocol. First we notice that $c = 2^{k-\ell-1}c'$ where $c' = (2^m r + r') + a + 2^\ell b - 2^{\ell+1} vb$, where v is set if and only if $(2^m r + r') + a$ overflows modulo 2^ℓ. It is easy to see that this implies that $c'_\ell = v \oplus b$, so we see that $v = c'_\ell \oplus b$, as calculated in the protocol.

On the other hand, we have that $(c' \bmod 2^\ell) = (2^m r + r') + a - 2^\ell v$, so $a \bmod 2^m =$

Deterministic Truncation over $\mathbb{F}_p$

Pre:

- Shares $[a] = [\mathsf{Rep}(\alpha)]$, integer $0 < m < \ell$.
- edaBit $([r]_M, [r]_2)$ of length $\ell - m + s$.
- edaBit $([r']_M, [r']_2)$ of length m.

Post: Shares $[y]$ where $y = \mathsf{Rep}\left(\lfloor \frac{\alpha}{2^m} \rfloor\right)$.

1. First the parties compute shares of $a \bmod 2^m$ as follows:
   a) Let $[b] = 2^{\ell-1} + [a]$;
   b) Call $c = \mathsf{open}([b] + 2^m[r] + [r'])$;
   c) The parties compute $[v]_2 = \mathsf{LT}\left((c_i)_{i=0}^{m-1}, ([r'_i]_2)_{i=0}^{m-1}\right)$;
   d) Convert $[v]_2 \mapsto [v]$.
   e) Let $[a \bmod 2^m] = [c \bmod 2^m] - [r'] + [v]2^m$.
2. Compute the truncated value using the formula as follows. Let $(2^m)^{-1}$ be the inverse of 2^m modulo p. Output $[y] = (2^m)^{-1} \cdot ([a] - [a \bmod 2^m])$.

Figure 3.15: Deterministic truncation over fields with share gap

$(c' \bmod 2^m) - r' + 2^m u$, where u is set if $(c' \bmod 2^m) < r'$. From this it can be obtained that $\lfloor (c' \bmod 2^\ell)/2^m \rfloor - r + 2^{\ell-m} = \lfloor a/2^m \rfloor + u$.

Remark 3.2. *The protocol we discussed above only works if $a \in [0, 2^\ell)$, that is, if the value α represented $\alpha \in [0, 2^{\ell-1})$. We can extend it to $\alpha \in [-2^{\ell-1}, 2^{\ell-1})$ by using the same trick as in the deterministic truncation: The truncation is called with $a + 2^{\ell-1}$ as input, and $2^{\ell-m-1}$ is subtracted from the output.*

Truncation over Fields.

We begin with a protocol, presented originally by Catrina and de Hoogh [Cd10], and optimize it with our edaBits. For this protocol we require a larger gap between the shares and the secret to be truncated, more precisely, it must hold that $p > 2^{\ell+s+1}$, where s is the statistical security parameter. The protocol is presented in Fig. 3.15.

To see the correctness of the protocol, begin by observing that because $p > 2^{\ell+s+1}$, and since $b \in [0, 2^\ell)$ the addition of b and $2^m r + r'$ does not overflow modulo p and therefore c is actually equal to $b + 2^m r + r'$, as integers. This preserves the privacy of b as $b \in [0, 2^\ell)$ and $2^m r + r'$ is uniformly random in $[0, 2^{\ell+s+1})$. Given this, it holds then that $(c \bmod 2^m) = (b \bmod 2^m) + (2^m r + r' \bmod 2^m) - v \cdot 2^m$, where $v \in \{0, 1\}$ is set if and only if $(b \bmod 2^m) + (r \bmod 2^m) \notin [0, 2^m)$. Now, observe that this condition triggers if and only if $c \bmod 2^m = \sum_{i=0}^{m-1} c_i 2^i$ is smaller than $r \bmod 2^m = \sum_{i=0}^{m-1} r_i 2^i$, so the bit v can be obtained by executing a (unsigned) binary less-than circuit as done by the protocol. We remark that for this step we use our optimized binary-shared bits, which provides an important optimization with respect to the protocol from Catrina et al.

Probabilistic truncation over $\mathbb{F}_p$

Pre:

- $\mathcal{F}_{\mathsf{ABB}}$ with $p > 2^{k+s+1}$.
- Shares $[a] = [\mathsf{Rep}(\alpha)]$, integer $0 < m < k$.
- edaBit $([r]_M, [r]_2)$ of length $k - m + s$.
- edaBit $([r']_M, [r']_2)$ of length m.

Post: Shares $[y]$ where $y \approx \mathsf{Rep}\left(\left\lfloor \frac{\alpha}{2^m} \right\rfloor\right)$.

1. Let $[b] = 2^{k-1} + [a]$;
2. Call $c = \mathsf{open}([b] + 2^m[r] + [r'])$;
3. Let $[d] = [c \bmod 2^m] - [r']$.
4. Output $[y] = (2^m)^{-1} \cdot ([a] - [d])$.

Figure 3.16: Probabilistic truncation over fields.

Taking into account that $(2^m r + r') \bmod 2^m = r'$, and also that $a \equiv b \bmod 2^{\ell-1}$, $(c \bmod 2^m) - r' + v \cdot 2^m$ is the same as $a \bmod 2^m$, we obtain that the first part of the protocol in which shares of $a \bmod 2^m$ are computed is correct. Finally, the ending step computes the formula for the truncation, which concludes the correctness analysis.

Probabilistic Truncation. The protocol above is not constant round, as it requires the computation of a less-than circuit on inputs of length m. It turns out that if one is willing to allow for some small error, a much more efficient protocol can be devised, as by Catrina and Saxena [CS10]. This protocol follows the same blueprint as the deterministic one, except that the computation of the overflow bit v is omitted. The description of the protocol can be found in Fig. 3.16. Following the analysis from the previous protocol, this implies that the value d computed in the protocol is $d = (a \bmod 2^m) - 2^m v$, so the final value computed is $(a - (a \bmod 2^m))/2^m + v$, which is the desired truncation, off by at most one bit. Furthermore, it is easy to see that the result is biased towards the nearest truncation.

Integer Comparison

Another important primitive that appears in many applications is integer comparison. In this case, two secret integers $[a]_M$ and $[b]_M$ are provided as input, and the goal is to compute shares of $\alpha \overset{?}{<} \beta$, where $a = \mathsf{Rep}(\alpha)$ and $b = \mathsf{Rep}(\beta)$.

As noticed by previous works (e.g. [Cd10, DEF+19]), this computation reduces to extracting the MSB from a shared integer as follows: If $\alpha, \beta \in [-2^{k-2}, 2^{k-2})$, then $\alpha - \beta = [-2^{k-1}, 2^{k-1})$, so $a - b = \mathsf{Rep}(\alpha - \beta)$ corresponds to the sign of $\alpha - \beta$, which is minus (i.e. the bit is 1) if and only if α is smaller than β.

To extract the MSB, we simply notice that $\mathsf{MSB}(\alpha) = -\left\lfloor \frac{\alpha}{2^{k-1}} \right\rfloor \bmod 2^k$, so this can be extracted with the protocols we have seen in the previous sections.

MSB extraction in $\mathbb{Z}_{2^k}$

Pre:

- $\mathcal{F}_{\mathsf{ABB}}$
- Input $[a]_{2^k}$ where $a \in [0, 2^\ell)$.
- Loose edaBit $([r]_{2^k}, [r]_2)$ of length ℓ

Post: $[y]_{2^k}$, where $y = \mathsf{LogShift}_{\ell-1}(a)$.

1. Call $c = \mathsf{open}\left(2^{k-\ell} \cdot ([a]_{2^k} + [r]_{2^k})\right)$
2. Compute $[v]_2 = \mathsf{LT}((c_i)_{i=k-\ell}^{k-2}, ([r_i]_2)_{i=0}^{\ell-1})$
3. Compute $[y]_2 = [v]_2 \oplus [r_{\ell-1}]_2 \oplus c_{k-1}$
4. Convert $[y]_2 \mapsto [y]_{2^k}$
5. Output $[y]_{2^k}$

Figure 3.17: Protocol to extract the most significant bit

Optimized Comparison with Power of Two Modulus. If the computation modulus is a power of two, the optimized protocol in Fig. 3.17 can be used. It uses the fact that an overflow can be erased by multiplication with an appropriate power of two.

3.7 Cost estimates

Δ-OT. This is a correlated OT where the sender's messages are of the form $(r_i, r_i \oplus \Delta)$, for random r_i and fixed $\Delta \in \{0,1\}^\lambda$.

Authenticated bits. Adding a MAC to a bit held by one party costs $(n-1)$ Δ-OTs (where n is the number of parties), plus $n-1$ bits of communication; the extra communication is avoided if the bit is random. To add a MAC to a bit that is *secret-shared* between all parties, multiply this by n.

Private triple. Authenticated triples $[a]_2, [b]_2, [c]_2$, where a single party P_i knows the values a, b, c. First P_i samples and authenticates a large batch of triples, then the parties check correctness with a cut-and-choose procedure.

Cost: Each authenticated triple needs 3 private authenticated bits, 2 of which are random.

Total cost per correct triple: $3B$ private authenticated bits and $B(n-1)$ bits of communication, where B is the bucket size in the cut-and-choose.

Triple. To create authenticated triples $[a]_2, [b]_2, [c]_2$, where a, b, c are secret-shared and known to nobody, we use one of the TinyOT triple generation protocols from [FKOS15, WRK17b].

Cost:

- Using [FKOS15]:
  - $3B^2$ random, secret-shared authenticated bits.
  - Additional $2B^2n(n-1)$ bits of communication to create the initial triples.
  - $2(B-1)Bn(n-1)$ bits for the openings in the sacrifice step
  - $(B-1)n(n-1)$ bits for the openings in the leakage removal step.
- Using [WRK17b]: $3B$ secret-shared authenticated bits plus $2B\lambda n(n-1)$ bits of communication

Private double sharings. Authenticated shares $([r_1]_2, \cdots, [r_k]_2)$ and $[r]_p$, where $r = \sum_{i=1}^{k} r_i \cdot 2^{i-1} \bmod p$ is known to a single party P_i. P_i creates a large batch of tuples, which are checked with cut-and-choose.

Cost: $(B-1) \cdot C_{\mathsf{add}}$ private triples, where C_{add} is the size of the binary circuit for addition modulo p. (Ignoring online phase cost of openings, should be negligible)

Global double sharings. Same as above, where r_i and r are secret-shared.

Cost: n times cost of private double sharings, plus $(n-1) \cdot C_{\mathsf{add}}$ secret-shared triples (to securely add up the n sets of private sharings).

Total cost of private double sharing.

$$\begin{aligned} C_{\mathsf{privDbl}} &\approx (B-1) \cdot C_{\mathsf{add}} \cdot C_{\mathsf{trip}}^{\mathsf{priv}} \\ &= (B-1) \cdot C_{\mathsf{add}} \cdot (3B \cdot (n-1) \cdot C_{\mathsf{OT}} + B(n-1)) \\ &= (n-1) \cdot B(B-1) \cdot C_{\mathsf{add}} \cdot (3C_{\mathsf{OT}} + 1) \end{aligned}$$

Total cost of global double sharing.

$$\begin{aligned} &\approx n \cdot C_{\mathsf{privDbl}} + (n-1) \cdot C_{\mathsf{add}} \cdot C_{\mathsf{trip}} \\ &= (n-1) \cdot C_{\mathsf{add}} \cdot (B(B-1) \cdot (3C_{\mathsf{OT}} + 1) + C_{\mathsf{trip}}) \end{aligned}$$

3.8 Applications and Benchmarks

Theoretical Cost

We present the theoretical costs of the different protocols in the paper, starting with the cost for producing Private and Global edaBits in terms of the different parameters.

Table 3.3 shows the main amortized costs for generating a Private and Global edaBit of length m. For Global edaBits, we assume have the required correct Private edaBits to start with, which is why number of Faulty edaBits needed is 0. B is the bucket size for the cut-and-choose procedure and n is the number of parties.

Table 3.4 shows the cost for two of our primitives from Section 3.6, namely comparison of m-bit numbers and truncation of an ℓ-bit number by m binary digits. For computation modulo a prime, there is also a statistical security parameter s.

Comparison in $\mathbb{Z}_{2^k}$ is our only application where it suffices to use loose edaBits (where the relation between the sets of shares only holds modulo 2^m, c.f. Section 3.4). This is because the arithmetic part of an edaBit is only used in the first step (the masking) but not at the end. Recall

Table 3.3: Amortized costs for generating 1 Private, and 1 Global edaBit. Costs for Global edaBits do not include the cost of the n additional sets of Private edaBits that are needed.

	Private edaBits		Global edaBits	
	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$
Faulty edaBits	B	B	0	$0\ (l-m+s,\ m)$
Faulty Triples	$(B-1)m$	$(B-1)m$	0	0
Secure Triples	0	0	$(\log n)(n-1)$	$(\log n)(n-1)$
daBits	0	$(B-1)$	0	$\log n$
Openings ($\mathbb{Z}_2$)	$(3m+1)(B-1)$	$(3m+1)(B-1)$	$(2m+2\log n)(n-1)$	$(2m+3\log n)(n-1)$
Openings ($\mathbb{Z}_M$)	$(B-1)$	$(B-1)$	0	0

Table 3.4: Cost of our primitives. Numbers in brackets indicate edaBit length.

	Comparison		Truncation	
	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$
Strict edaBits	0	$2\ \{m+1,\ s+1\}$	$2\ \{l-m,\ m\}$	$2\ \{l-m+s,\ m\}$
Loose edaBits	$1\ \{m+1\}$	0	0	0
classic daBits	1	1	2	1
Online ANDs	$\sim 2m$	$\sim 2m$	$\sim 2m$	$\sim 2k$

that the truncation protocols always use the arithmetic part of an edaBit twice, once before opening and once to compute an intermediate or the final result. Using a loose edaBit would clearly distort the result. With comparison on the other hand, an edaBit is only used to facilitate the conversion to binary computation, after which the result is converted back to arithmetic computation using a classic daBit.

Implementation Results

We have implemented our approach in a range of domains and security models, and we have run the generation of a million edaBits of length 64 on AWS `c5.9xlarge` with the minimal number of parties required by the security model (two for dishonest majority and three for honest majority). Table 3.5 shows the throughput for various security models and computation domains, and Table 3.6 does so for communication. In the prime field case, we use $\log p \approx 128$ to allow additional room needed for comparisons, while for arithmetic mod 2^k we use $k = 64$. For computation modulo a prime with dishonest majority, we present figures for arithmetic computation both using oblivious transfer (OT) [KOS16] and LWE-based semi-homomorphic encryption (HE) [KPR18]. Note that the binary computation is always based on oblivious transfer for dishonest majority and that all our results include all consumable preprocessing such as multiplication triples but not one-off costs such as key generation.

We have also implemented 63-bit[3] comparison using edaBits, only daBits, and neither, and we have run one million comparisons in parallel again on AWS `c5.9xlarge`. Table 3.7 shows the throughput for our various security models and computation domains, and Table 3.8 does so

[3]Comparison in secure computation is generally implemented by extracting the most significant bit of difference. This means that 63-bit is the highest accuracy achievable in computation modulo 2^{64}, which the natural modulus on current 64-bit platforms.

Table 3.5: Number of edaBits generated (in 1000s) per second in various settings

		Domain	Strict edaBits	Loose edaBits
Dishonest maj.	Malicious	2^k (OT)	4.6	7.3
		p (OT)	3.6	4.2
		p (HE)	2.7	3.4
	Semi-hon.	2^k (OT)	456.7	922.5
		p (OT)	228.0	892.6
		p (HE)	470.5	905.6
Honest maj.	Malicious	2^k	191.5	205.8
		p	156.6	162.1
	Semi-hon.	2^k	2032.1	7180.0
		p	1367.7	4934.3

Table 3.6: Communication per edaBit (in kbit) in various settings

		Domain	Strict edaBits	Loose edaBits
Dishonest maj.	Malicious	2^k (OT)	1335.5	480.2
		p (OT)	1936.9	1473.2
		p (HE)	940.8	779.7
	Semi-hon.	2^k (OT)	22.5	9.6
		p (OT)	43.9	9.6
		p (HE)	11.8	9.6
Honest maj.	Malicious	2^k	5.6	3.7
		p	7.6	6.4
	Semi-hon.	2^k	0.3	0.2
		p	0.5	0.2

for communication. Note that the arithmetic baseline uses either the protocol of Catrina and de Hoogh [Cd10] ($\mathbb{F}_p$) or the variant by Dalskov et al. [DEK19] ($\mathbb{Z}_{2^k}$).

Our results highlight the advantage of our approach over using only daBits. The biggest improvement comes in the dishonest majority with semi-honest security model. For the dishonest majority aspect, this is most likely because there is a great gap in the cost between multiplications and inputs (the latter is used extensively to generate edaBits). For the semi-honest security aspect, note that our approach for malicious security involves a cascade of sacrificing because the edaBit sacrifice involves binary computation, which in turn involves further sacrifice of AND triples. Finally, the improvement in communication is generally larger than the improvement in wall clock time. We estimate that this is due to the fact that switching to binary computation clearly reduces communication but increases the computational complexity.

Comparison to Previous Works

Dishonest majority. The authors of HyCC [BDK+18] report figures for biometric matching with semi-honest two-party computation in ABY [DSZ15] and HyCC. The algorithm essentially computes the minimum over a list of small-dimensional Euclidean distances. The aforementioned authors report figures in LAN (1Gbps) and artificial WAN settings of two machines with four-core

Table 3.7: Number of comparisons (in 1000s) per second in various settings

		Domain	Arithm.	daBits	edaBits
Dishonest maj.	Malicious	2^k (OT)	0.5	1.2	4.4
		p (OT)	0.3	0.3	1.6
		p (HE)	0.6	0.7	2.0
	Semi-hon.	2^k (OT)	5.2	14.4	275.6
		p (OT)	1.6	3.3	79.7
		p (HE)	5.9	12.8	170.6
Honest maj.	Malicious	2^k	76.4	119.2	170.4
		p	66.9	78.3	80.1
	Semi-hon.	2^k	500.6	1007.7	1607.6
		p	157.8	277.1	457.6

Table 3.8: Communication per comparison (in kbit) in various settings

		Domain	Arithm.	daBits	edaBits
Dishonest maj.	Malicious	2^k (OT)	21737.7	9058.6	1310.5
		p (OT)	40108.5	34019.1	4783.3
		p (HE)	3020.5	3210.9	1584.8
	Semi-hon.	2^k (OT)	2283.0	830.2	39.0
		p (OT)	7353.1	3503.0	134.9
		p (HE)	411.6	219.1	38.7
Honest maj.	Malicious	2^k	63.4	27.8	5.4
		p	94.3	85.0	19.9
	Semi-hon.	2^k	14.5	7.1	0.4
		p	37.4	23.1	1.4

i7 processors. For a fair comparison, we have run our implementation using one thread limiting the bandwidth and latency accordingly. Table 3.9 shows that our results improves on the time in the LAN setting and on communication generally as well as on the in the WAN setting for larger instances compared to their A+B setting (without garbled circuits). The WAN setting is less favorable to our solution because it is purely based on secret sharing and we have not particularly optimized the number of rounds.

Honest majority (three parties, one semi-honest corruption). Our approach is not directly comparable to ABY3 by Mohassel and Rindal [MR18] because they use the specifics of replicated secret sharing for the conversion. We do note however that their approach of restricting binary circuits to the binary domain is comparable to our solution, and that they use the same secret sharing schemes as us in the 2^k domain. We compare their results with our approach applied to logistic regression. Their software implementation [MR19] runs all parties on the same host without communication encryption. For a fair comparison, we have run their software as well as ours in the same setting on the same desktop machine with an i7 processor. In our software, we use the special truncation according to Dalskov et al. [DEK19] and either edaBits or bit decomposition as in the work above for comparison. The comparison in turn is used for a

Table 3.9: Overall time and communication for biometric matching

		LAN (s)	WAN (s)	Comm. (MB)
$n = 1000$	ABY/HyCC (A+Y)	0.22	2.5	9.5
	ABY/HyCC (A+B)	0.22	6.1	10.6
	Ours	0.12	8.3	7.4
$n = 4096$	ABY/HyCC (A+Y)	0.63	6.6	40.4
	ABY/HyCC (A+B)	0.72	13.6	43.6
	Ours	0.48	12.6	29.1
$n = 13684$	ABY/HyCC (A+Y)	3.66	17.5	138.0
	ABY/HyCC (A+B)	5.4	26.2	190.8
	Ours	2.00	22.9	111.8

Dimension	Batch size	ABY3 [MR18]	Ours (ABY3 comp.)	Ours (edaBits)
10	128	1495	1801	1671
	256	1402	1407	1230
	512	1229	1014	827
	1024	976	656	479
100	128	1303	1372	1269
	256	1064	988	904
	512	732	657	560
	1024	349	387	316
1000	128	327	436	422
	256	148	284	271
	512	74	167	159
	1024	35	90	84

Table 3.10: Iterations per second for logistic regression

piece-wise approximation of the sigmoid function. Table 3.10 shows that edaBit-based comparison generally comes close to ABY3's bit decomposition.

daBits. Aly et al. [AOR+19] report figures for daBit generation with dishonest majority and malicious security in eight threads over a 10 Gbps network. For two-party computation using homomorphic-encryption, they achieve 2150 daBits per second at a communication cost of 94 kbit per daBit. In a comparable setting, we found that our protocol produces 12292 daBits per second requiring a communication cost of 32 kbit. Note however that Aly et al. use somewhat homomorphic encryption while our implementation is based on cheaper semi-homomorphic encryption.

Convolutional Neural Networks. We also apply our techniques to the convolutional neural networks.Dalskov et al. [DEK19] present an implementation for deep learning inference. We have adapted their implementation to our setting and present a comparison for the simplest network (MobileNet V1 0.25_128) in Table 3.11. It shows that edaBits reduce the communication and

		Domain		Time (s)	Comm. (GB)
Dish. maj.	Mal.	2^k (OT)	[DEK19]	1264.9	1748.4
			Ours	455.3	561.9
		p (HE)	[DEK19]	1377.8	282.4
			Ours	552.9	299.9
	S-h.	2^k (OT)	[DEK19]	139.5	199.2
			Ours	23.8	32.4
		p (HE)	[DEK19]	129.1	37.1
			Ours	22.4	6.8
Hon. maj.	Mal.	2^k	[DEK19]	5.3	2.5
			Ours	3.4	2.2
		p	[DEK19]	9.0	8.7
			Ours	8.3	4.6
	S-h.	2^k	[DEK19]	0.2	0.1
			Ours	0.3	0.1
		p	[DEK19]	3.3	3.4
			Ours	2.2	0.3

Table 3.11: Time and communication for MobileNet inference

time in most security models. The only exception is semi-honest honest-majority computation modulo 2^k, where Dalskov et al. use the conversion by Mohassel et al. [MR18], which has similar properties to our approach. The figures for malicious protocols have been generated using bucket size four because the batches would otherwise far exceed the required edaBits.

Chapter 4

Le Mans: MPC for Dynamic Participants

The contents of this chapter have been taken from the paper [RS22]. The only modifications made are to move the appendix content into the main body for better readability.

4.1 Introduction

Secure multi-party computation (MPC) allows a set of parties to jointly compute a function on their inputs, while preserving privacy, that is, not revealing anything more about the inputs than can be deduced from the output of the function. MPC can be applied in a wide range of situations, including secure aggregation, private training or evaluation of machine learning models, threshold signing and more.

Most MPC protocols work under the assumption that the set of parties involved in the computation is fixed throughout the protocol. Although committee-based MPC and player-replaceability schemes have existed for a while, recently more practically oriented models have been proposed such as Fluid MPC [CGG$^+$21] and YOSO [GHK$^+$21]. These models support protocols with a *dynamically evolving* set of parties, where participants can join and leave the computation as desired, without interrupting the protocol. This enables a more flexible model, where parties can sign up to contribute their resources towards a large-scale, distributed computation, without having to commit for the duration of the entire protocol. This is particularly important for large-scale, long-running tasks such as complex scientific computations, such as Folding@home. In the *maximally fluid* setting, this concept is pushed to the limit, where each participant is only required to sign up for *a single round* of the protocol. This gives the most possible flexibility for any server who may wish to participate.

The YOSO (you only speak once) paradigm [GHK$^+$21] also considers maximally fluid MPC protocols, with some differences in the model. Unlike Fluid MPC, they separately study the role assignment problem, where they show how to leverage a blockchain to randomly assign the committee of parties who will take part in each round. With their mechanism, the identity of any member of the current committee is only revealed after they have published their message. This allows for much stronger security guarantees, since an adversary has no way to identify which servers are involved in the computation — and hence who to corrupt — until the role played by the server has already been terminated.

Both of these works give information-theoretically secure protocols in the *honest majority* setting, where in any given round of the protocol, the majority of the computing parties should be honest. Fluid MPC achieves security with abort, where a malicious party can prevent the protocol from terminating, while YOSO achieves the stronger notion of guaranteed output delivery (but is less efficient).

Our Contributions

In this work, we study MPC with dynamically evolving parties in the *dishonest majority* setting. This gives much stronger security guarantees, since we only require that in any given round of the computation, there is at least one honest party taking part. However, it is also more challenging than honest majority. We now elaborate on our contributions and some technical background.

The challenge of fluidity and dishonest majority.

In the dishonest majority setting, most practical MPC protocols are based on authenticated secret-sharing using information-theoretic MACs, such as in the SPDZ [DPSZ12] or BDOZ [BDOZ11] protocols. These protocols rely on a preprocessing phase, using more expensive, "public-key" style cryptography, to generate a large amount of correlated randomness that is consumed in a lightweight online phase. Unfortunately, this means that each party has to maintain a *large state* (the correlated randomness), the size of which grows linearly with the complexity of the function being computed. This is problematic for achieving Fluid MPC, since when changing from one committee of parties to another, the natural approach is to securely transfer the entire state to the new committee. Ideally, we want this state transfer process to be *independent* of the function being computed, to avoid the communication complexity blowing up.

Key Tool: Universal Preprocessing for Dynamic Parties.

Before aiming for Fluid MPC, we look at a simpler model which allows just a single change in the set of computing parties during the protocol. We consider a *universal preprocessing* phase, where all of the parties $P_1, \ldots, P_n$ who may wish to be involved in the computation must take part. Later, any subset of the n parties can get together and run a fast, online protocol, without having to interact with anybody else. We assume the inputs to the protocol are provided by the online subset of parties (though with standard techniques such as [DDN$^+$16], we can also support inputs from external parties).

Recall that in SPDZ, the parties need to preprocess authenticated multiplication triples, denoted $[\![a]\!], [\![b]\!], [\![c]\!]$, where a and b are secret, random finite field elements and $c = a \cdot b$. These values are secret-shared with MACs, given by

$$[\![x]\!] := (x^i, m^i, \Delta^i)_{i \in [n]}$$

where party P_i has the share Δ^i of the global MAC key $\Delta = \sum \Delta^i$, and also the shares x^i, m^i, satisfying $x = \sum x^i$ and $x \cdot \Delta = \sum m^i$ over the field.

Instead of producing fully authenticated triples like this, we produce a weaker form of *partial triple*, where c is unauthenticated, and not fully computed: every pair of parties (P_i, P_j) will get a two-party additive sharing of $a^i \cdot b^j$. This suffices to reconstruct a share c^i, by adding up P_i's relevant sharings of $a^i b^j$, together with $a^i b^i$.

Importantly, this also enables *any subset* of parties $\mathcal{P} \subset [n]$ to obtain a triple, by restricting to the shares a^i, b^i for $i \in \mathcal{P}$, and summing up the relevant shares of the products to get a c^i for this committee. A similar trick also works to get the MACs on a and b, since each MAC is just a secret-shared product with the fixed key Δ. Therefore, it's enough to give out two-party shares of $a^i \Delta^j$ and $b^i \cdot \Delta^j$ for every $i \neq j$.

We show how to realize this type of preprocessing using simple, pairwise correlations between every pair of parties, in the form of oblivious linear function evaluation (OLE) and vector-OLE. We ensure correctness of the authenticated $[\![a]\!], [\![b]\!]$ shares using a consistency check, which we formalize via a multi-party vector-OLE functionality. However, our protocol does not guarantee correctness of the shares of cross-products $a^i \cdot b^j$. We therefore model these errors via adversarial influence in the preprocessing functionality.

PCG-Friendliness.

An important feature of our preprocessing protocol is that it is *PCG-friendly*, meaning that it can be implemented using *pseudorandom correlation generators* (PCGs) [BCG$^+$19b]. A PCG allows two parties to take a pair of short, correlated seeds, and expand them to produce a much larger quantity of correlated randomness. There are efficient PCGs for vector-OLE, based on variants of the LPN assumption [BCGI18, BCG$^+$19a, WYKW20], and for OLE under a variant of ring-LPN [BCG$^+$20]. By supporting PCGs in our preprocessing, we obtain communication and storage complexities as small as $O(n \log|C|)$ field elements per party, for an arithmetic circuit C. Prior to our work, we stress that even with a statically chosen online phase, there was no practical, multi-party SPDZ-like protocol[1] that could support a preprocessing phase with this feature with good concrete efficiency — ours is the first protocol to support this "silent" feature. In recent, concurrent work [BGIN22], another MPC protocol with sublinear preprocessing was given. Their preprocessing protocol also relies on PCGs, but scales with the square root of the circuit size rather than logarithmically. However, their online phase communication is slightly better than ours, and the communication of their preprocessing phase scales better with the number of parties.

Dynamic Variant of SPDZ Online Phase.

One issue with our universal preprocessing is that, since the c terms of triples are not authenticated, we cannot use the same online phase as SPDZ. Instead, we modify the online phase so that in each multiplication, we first authenticate c before using a triple to multiply. Since a malicious party may have introduced errors in c, we then need to add a *verification phase*, to check the multiplications are correct. We do this following the approach of Chida et al. [CGH$^+$18] (also used by the honest majority Fluid compiler of [CGG$^+$21]). Here, as well as computing the circuit, the parties compute a randomised version of the circuit, where each wire value has been multiplied by a secret, random value $r \in \mathbb{F}_p$. At the end of the computation, the parties run a batch verification process to check consistency of the two computations. We show that this guarantees our protocol is correct, even with our weaker preprocessing protocol which allows malicious parties to introduce special types of errors into c.

Overall, the communication cost of our dynamic online protocol is only 8 field elements on top of the SPDZ online phase [DPSZ12, DKL$^+$13], which costs 4 elements per party. However, this comes with the benefits of (1) a dynamically chosen online committee, and (2) a PCG-friendly preprocessing phase, where each party's communication and storage complexity is $O(n \log|C|)$, instead of $O(|C|)$ storage and $O(n|C|)$ communication for standard SPDZ preprocessing. Note that after locally expanding the PCG seeds, the preprocessing material for our dynamic and

[1]In the two party setting, an efficient PCG-based SPDZ preprocessing protocol was given in [BCG$^+$19b].

fluid protocols has size $O(n|C|)$ per party, which is n times larger than SPDZ. However, once the online committee is known in Dynamic SPDZ, this can be compressed down to $O(|C|)$.

Maximally Fluid Online Phase.

We now turn to the harder task of obtaining an online phase where the set of computing parties can dynamically change. We focus on the most challenging goal of *maximal fluidity*, where in each round, a different committee can sign up to receive one round of messages from the previous committee, before sending one round of messages and going offline.

This brings additional obstacles when it comes to preprocessing data, as well as verifying MACs on opened values during the online protocol. Since the MAC key of a committee is determined by the sum of the MAC keys of the parties in it, different committees will have different MAC keys. The issue with this is that, even though our universal preprocessing allows any committee to obtain a multiplication triple, these triples end up being authenticated under different MAC keys, depending on the committee. Hence, re-sharing state from one committee to another will lead to values that are authenticated under a different MAC key.

As a first attempt to deal with this problem, one could have the current committee, $\mathcal{P}_{\mathsf{curr}}$, securely *reshare* the current state of intermediate computation values, including their MAC key $\Delta_{\mathcal{P}_{\mathsf{curr}}}$, to the next committee, $\mathcal{P}_{\mathsf{next}}$. To proceed further, however, $\mathcal{P}_{\mathsf{next}}$ will need authenticated triples under the same MAC key. Our preprocessing phase, on the other hand, only allows them to obtain triples under a different key $\Delta_{\mathcal{P}_{\mathsf{next}}}$. To avoid this issue, $\mathcal{P}_{\mathsf{curr}}$ would instead have to reshare *all of* the triples needed for the rest of the circuit evaluation, after which, $\mathcal{P}_{\mathsf{next}}$ would use some of these, reshare to the next committee and so on. This incurs a huge blow up in communication cost, which we would like to avoid.

Our method for dealing with this is a secure *key-switching* procedure, which allows $\mathcal{P}_{\mathsf{curr}}$ to transfer a shared $[\![x]\!]$ to $\mathcal{P}_{\mathsf{next}}$ in a single round, while switching to $\mathcal{P}_{\mathsf{next}}$'s MAC key. Another constraint we have from the model is that $\mathcal{P}_{\mathsf{next}}$ cannot send any messages to $\mathcal{P}_{\mathsf{curr}}$. At first glance, it may seem impossible, since $\mathcal{P}_{\mathsf{curr}}$ should not have any information on the next key. However, we show that by leveraging the power of our universal preprocessing, key-switching can be done with just a single set of messages from $\mathcal{P}_{\mathsf{curr}}$ to $\mathcal{P}_{\mathsf{next}}$.

In addition to securely switching keys, another challenge in our maximally fluid protocol is how to check MACs on opened values. We cannot use the batched MAC check from SPDZ, since this involves storing a large state, which has to be passed around until the end of the protocol. Instead, we modify this to an incremental procedure, where only a constant-sized state needs to be transferred in each round. We adopt a similar incremental protocol to verify multiplications, where, as in our Dynamic SPDZ protocol, we use the same randomised circuit idea as [CGH+18].

Related Work

Bracha [Bra85] introduced the idea of using committees in distributed protocols with a large number of parties, which has been used in a number of MPC protocols since. One recent example is [GSY21], which constructs committee-based MPC when up to 1/3 of the parties may be corrupt, achieving a construction that scales to hundreds of thousands of parties. Although part of their protocol is based on SPDZ, they do not support the notion of a dynamically chosen subset of parties from the preprocessing set carrying out the online computation. Concretely, their online phase for circuit evaluation costs 7x higher than SPDZ, whereas we estimate that we only suffer a 3x overhead. A detailed analysis of the costs is provided in Section 4.6.

Functionality $\mathcal{F}_{\mathsf{Rand}}$

The functionality runs between a set of parties $\mathcal{P}$ and an adversary $\mathcal{A}$.
Upon receiving a description of a domain $\mathbb{F}_{p^r}^m$ from every party in $\mathcal{P}$, uniformly sample $(x_1, \ldots, x_m) \leftarrow_R \mathbb{F}_{p^r}^m$ and send this to $\mathcal{A}$. If $\mathcal{A}$ responds with $\mathsf{Deliver}$, send $x_1, \ldots, x_m$ to all parties and terminate. Otherwise, if $\mathcal{A}$ sends Abort, send Abort to all parties and terminate.

Figure 4.1: Ideal functionality for coin tossing

Another relevant work is [SSW17], which outsources SPDZ preprocessing to an external set of parties. However, unlike our protocol, this requires resharing the entire preprocessing data from the external set to the online committee. We avoid this in Dynamic SPDZ, by relying on our universal preprocessing.

The area of proactive security has long considered the notion of an adversary who can corrupt different parties throughout the computation. These works typically use a proactive secret sharing scheme, where secrets are maintained by an ever-changing set of parties. Works such as [HJKY95, MZW$^+$19] show security in the presence of a mobile adversary that can corrupt and uncorrupt parties at different points in the protocol. More recently, [BGG$^+$20, GKM$^+$20] construct secret-sharing protocols for the case of honest majority with active security. The model used in these papers also splits the work done by each committee into two parts, one used to do the computation with parties interacting only within the committee, and one used to perform a secure state-transfer to the committee that comes after them. The primary difference between Fluid SPDZ and proactive MPC is the motivation and the behaviour of the adversary. In proactive schemes, the adversary typically operates with a “corruption budget" that limits the adversary from being able to corrupt parties arbitrarily. We do not make such an assumption, and our motivation primarily comes from giving parties in a computation the ability to drop in and out, while minimising the minimum number of rounds they have to stay on for. In addition, we try to achieve a small *state complexity*, so that switching committees is not communication intensive.

4.2 Preliminaries and Security Model

Preliminaries

We use κ as the security parameter and ρ as the statistical security parameter. Bold letters such as $\boldsymbol{a}$ are used to indicate vectors, and $\boldsymbol{a}[i]$ refers to the i-th element of the vector. We write $[a, b]$ to denote the set of natural numbers $\{a, \ldots, b\}$ and $[a, b) = \{a, \ldots, b-1\}$.

Additional Functionalities. We make use of some standard functionalities such as a functionality for oblivious transfer $\mathcal{F}_{\mathsf{OT}}$ (Fig. 4.4), coin-tossing $\mathcal{F}_{\mathsf{Rand}}$ (Fig. 4.1), commitment $\mathcal{F}_{\mathsf{Commit}}$ (Fig. 4.3), and a weak equality test $\mathcal{F}_{\mathsf{EQ}}$ (Fig. 4.2), that checks equality of two private inputs, while always revealing one party's input to the adversary.

Modelling Fluid MPC in Dishonest Majority

The remainder of this subsection covers definitions pertaining to the Fluid model. Computation broadly proceeds in 4 phases – preprocessing, input, execution, and output. This is similar to that of Fluid MPC [CGG$^+$21], with the addition of a preprocessing phase, which is used to generate

Functionality $\mathcal{F}_{\mathsf{EQ}}$

This functionality receives a value V_A from P_A and V_B from P_B, checks if $V_A = V_B$, and reveals P_A's input to P_B.
Equality Check: On input (EQ, V_i) from P_i for $i \in [A, B]$:

1. Send V_A to P_B.
2. If P_B is honest, output $\mathsf{success}$ or fail depending on $V_A \overset{?}{=} V_B$ to P_A.
3. If P_B is corrupted, output to P_A whatever P_B sends.

Figure 4.2: Functionality to for a weak equality check

Functionality $\mathcal{F}_{\mathsf{Commit}}$

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

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

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

Figure 4.3: Ideal functionality for commitments

Functionality $\mathcal{F}_{\mathsf{OT}}$

On receiving (m_0, m_1) from P_A (sender), where $|m_0| = |m_1|$, and $b \in \{0, 1\}$ from P_B (receiver), output m_b to P_B.

Figure 4.4: Functionality to for oblivious transfer

data-independent information in the form of multiplication triples, to be used in the execution phase. In the preprocessing phase, we require all parties who wish to take part in the computation at some later point to be active, and after this they may go offline. The execution phase proceeds in epochs, where each epoch runs among a fixed set of parties, or committee. An epoch contains two parts, the *computation phase*, where the committee performs some computation, followed by a *hand-off phase*, used to securely transfer the current state to the next committee.

Fluidity. The computation phase of each epoch may take several rounds of interaction. Fluidity is defined as the minimum number of rounds in any given epoch of the execution phase. We say that a protocol achieves *maximal fluidity* if the epoch only lasts for one round. This means each server in the committee does some local computation, before sending a single message to the next committee in the hand-off phase. In the input and output phases, we do not measure fluidity, instead, the committee may interact for several rounds to share inputs or reconstruct the outputs.

A server is said to be “active" in the computation if it either performs computations or sends and/or receives messages. Therefore, a server participating in epoch i is active starting from the hand-off phase of epoch $i - 1$, until the end of the hand-off phase of epoch i.

Committee formation. The committees used in each epoch may be either fixed ahead of time, or chosen on-the-fly throughout the computation. Fixing them ahead of time can be useful, for instance, in a volunteer sign-up based model, where servers can volunteer to participate in any epoch, and stay on for any number of epochs depending on their resource constraints. On the other hand, choosing committees on-the-fly may be desirable in settings closer to the YOSO model [GHK$^+$21], where a role-assignment mechanism is used to ensure that the next committee is only revealed at the last possible moment.

In this work, we do not distinguish between these two cases, and instead simply require that during the hand-off phase of epoch i, the current committee, denoted $\mathcal{P}_i$, knows the identities of the parties in the next committee $\mathcal{P}_{i+1}$. We make no assumptions or restrictions about the overlap between committees. As in [CGG$^+$21], the formation process can be modelled with an ideal functionality that samples and broadcasts committees according to the desired mechanism.

Corruption. Our model allows all-but-one of the servers who are active at the start of any given epoch to be corrupted, where the set of corrupt parties is fixed at the beginning of the epoch. Formally, this corresponds to an R-*adaptive adversary* from [CGG$^+$21]. Here, at the beginning of epoch i with committee $\mathcal{P}_i$, the adversary may adaptively choose a set of servers in $\mathcal{P}_i$ to be corrupted, and then learns the entire state of each corrupted server in any prior epochs. For the duration of epoch i, this set of corrupted parties is then fixed and cannot change. To rule out the adversary learning information on prior epochs, a server S may be corrupted in epoch i only if this does not lead to any prior epoch j with committee $\mathcal{P}_j$ becoming entirely corrupt.

We use this model for the online phase of our fluid MPC protocol. Note that for our dynamic SPDZ protocol, where the online committee does not change, this corresponds to the more common notion of static security. In the preprocessing phase for both dynamic SPDZ and our fluid MPC protocol, we have only proven security against a static adversary. While for fluid MPC, we would ideally also like the preprocessing to be adaptively secure, this is particularly challenging in the dishonest majority setting, and is known to imply strong primitives like non-committing encryption. In fact, since no practical adaptively secure preprocessing protocols are even known for the standard SPDZ protocol [DPSZ12], we view this as an interesting open problem.

Security Model

To model fluid MPC, we adopt the arithmetic black box model (ABB), which is an ideal functionality $\mathcal{F}_{\mathsf{ABB}}$ in the universal composability framework [Can01]. The functionality allows for a set of parties $P_1, \dots, P_n$ to input their values, perform computations on them, and receive the outputs. The functionality is parameterised by a finite field $\mathbb{F}_p$, and supports native operations of addition and multiplication in the field.

We instantiate $\mathcal{F}_{\mathsf{ABB}}$ with the Dynamic SPDZ protocol ($\Pi_{\mathsf{SPDZ\text{-}Online}}$), which uses a preprocessing phase between a set of parties, and supports a dynamically chosen subset to perform the online phase. The preprocessing phase is used to set up partially authenticated, partially formed triples using pairwise MACs similar to BDOZ [BDOZ11] and TinyOT [HSS17]. We adapt the vector OLE from Wolverine [WYKW21b], and PCGs from [BCG$^+$19a] and use them to form the partial triples.

To model Fluid MPC, we modify $\mathcal{F}_{\mathsf{ABB}}$ to support computations with dynamic committees, as functionality $\mathcal{F}_{\mathsf{DABB}}$ in Fig. 4.5. The main difference is that now, the functionality keeps track of the currently active committee in a variable $\mathcal{P}_{\mathsf{curr}}$. In operations which are part of the execution phase, where the committee may change, the functionality receives the identity of the next committee from the currently active parties (if it receives inconsistent inputs, we assume

Functionality $\mathcal{F}_{\mathsf{DABB}}$

Parameters: Finite field $\mathbb{F}_p$, and set of parties $\mathcal{P}_{\mathsf{main}} = \{P_1, \dots, P_n\}$. The functionality assumes all parties have agreed upon public identifiers id_x, for each variable x used in the computation. For a vector $\boldsymbol{x} = (x_1, \dots, x_m)$, we write $\mathsf{id}_{\boldsymbol{x}} = (\mathsf{id}_{x_1}, \dots, \mathsf{id}_{x_m})$.

Initialise: On input $(\mathsf{Init}, \mathcal{P}_{\mathsf{curr}})$ from P_i, for $i \in [1, n]$, where each P_i sends the same set $\mathcal{P}_{\mathsf{curr}} \subset \mathcal{P}_{\mathsf{main}}$, initialise $\mathcal{P}_{\mathsf{curr}}$ as the first active committee.

Input: On input $(\mathsf{Input}, \mathsf{id}_x, x)$ from some $P_i \in \mathcal{P}_{\mathsf{main}}$, and $(\mathsf{Input}, \mathsf{id}_x)$ from all parties in $\mathcal{P}_{\mathsf{curr}}$, store the pair (id_x, x).

Add: On input $(\mathsf{Add}, \mathsf{id}_{\boldsymbol{z}}, \mathsf{id}_{\boldsymbol{x}}, \mathsf{id}_{\boldsymbol{y}})$ from P_i, for every $P_i \in \mathcal{P}_{\mathsf{curr}}$, compute $\boldsymbol{z} = \boldsymbol{x} + \boldsymbol{y}$ and store $(\mathsf{id}_{\boldsymbol{z}}, \boldsymbol{z})$.

Batch Multiply: On input $(\mathsf{Mult}, \mathcal{P}_{\mathsf{next}}, \mathsf{id}_{\boldsymbol{z}}, \mathsf{id}_{\boldsymbol{x}}, \mathsf{id}_{\boldsymbol{y}})$ from every $P_i \in \mathcal{P}_{\mathsf{curr}}$:

- Compute $\boldsymbol{z} = \boldsymbol{x} * \boldsymbol{y}$.
- Update $\mathcal{P}_{\mathsf{curr}} := \mathcal{P}_{\mathsf{next}}$.
- Wait to receive a message $(\mathsf{MultFinish}, \mathcal{P}'_{\mathsf{next}})$ from every $P_i \in \mathcal{P}_{\mathsf{curr}}$. Then, store the batch of products $(\mathsf{id}_{\boldsymbol{z}}, \boldsymbol{z})$ and update $\mathcal{P}_{\mathsf{curr}} := \mathcal{P}'_{\mathsf{next}}$.

Output: On input $(\mathsf{Output}, \mathsf{id}_{\boldsymbol{z}})$ from every $P_i \in \mathcal{P}_{\mathsf{curr}}$, where $\mathsf{id}_{\boldsymbol{z}}$ has been stored previously, retrieve $(\mathsf{id}_{\boldsymbol{z}}, \boldsymbol{z})$ and send it to the adversary. Wait for input from the adversary, if it is $\mathsf{Deliver}$, send the output to every $P_i \in \mathcal{P}_{\mathsf{curr}}$. Otherwise, abort.

Figure 4.5: Functionality for a dynamic arithmetic black box

it aborts). In our protocol, the **Batch Multiply** command is the only part of the execution phase with interaction, so this is where any changes in committee might take place. We have $\mathcal{P}_{\mathsf{curr}}$ provide the next committee $\mathcal{P}_{\mathsf{next}}$ as input, and then wait for another message from $\mathcal{P}_{\mathsf{next}}$, who will provide a subsequent committee $\mathcal{P}'_{\mathsf{next}}$. This is because our multiplication protocol takes place over two rounds, so it inherently allows up to two committee changes whenever it is called (if we want to support maximal fluidity).

In practice, with our protocol it is possible to interleave multiplications, so that a new multiplication can be started before the old one has finished (reducing round complexity). However, for simplicity, we do not model this in $\mathcal{F}_{\mathsf{DABB}}$.

We instantiate $\mathcal{F}_{\mathsf{DABB}}$ with a Fluid Online ($\Pi_{\mathsf{Fluid\text{-}Online}}$) protocol. It extends the model of Fluid MPC [CGG$^+$21] which only works for the honest majority case, to the dishonest majority setting with active security. It uses the same preprocessing phase as Dynamic SPDZ, but the online phase supports committees switching. Parties can leave the computation by securely transferring their state to the subsequent committee, and rejoin the computation at a later point.

4.3 Universal Preprocessing for Dynamic Committees

In this section, we present the preprocessing phase used in our two online protocols. Our main design goals are (1) to allow a flexible and dynamic choice of participants during the online phase, and (2) to obtain a silent preprocessing phase, where the storage and communication complexities are (almost) independent of the function being computed. The section is organised in a top-down manner, where we start by describing an ideal preprocessing functionality, and then gradually explain our protocol for realising it.

Overview.

In this section, we focus on realising $\mathcal{F}_{\mathsf{Prep}}$, using variants of oblivious linear function evaluation (OLE), as well as how to realise a multi-party variant of vector-OLE ($\mathcal{F}_{\mathsf{nVOLE}}$).

Preprocessing Functionality

Let $\mathcal{P}_{\mathsf{main}} = \{P_1, \dots, P_n\}$ be the set of all parties who may want to participate in the online phase.

Authenticated Secret Sharing.

For the preprocessing, we use two kinds of secret sharing. $[x]$ denotes that $x \in \mathbb{F}_p$ is additively shared between the parties, that is, $x = x^1 + \ldots + x^n$ where P_i holds x^i. We also use pairwise authenticated shares, indicated by $\langle x \rangle$. Here, in addition to an additive share of x, each party holds an information-theoretic MAC on their share with every other party, who holds a corresponding MAC key. The MAC of P_i's share x^i under P_j's key is defined as $M_j^i = K_i^j + \Delta^j \cdot x^i$, where P_i holds the MAC M_j^i and P_j holds the local key K_i^j as well as the global key Δ^j (which is fixed for all MACs). While the shares x^i lie over the field $\mathbb{F}_p$, we allow MAC keys and MACs to be in an extension field $\mathbb{F}_{p^r}$, giving a forgery probability of p^{-r}, in case p is not large enough for the desired statistical security level.

If x is only shared between a smaller committee $\mathcal{P}_C \subset \mathcal{P}_{main}$, we write $[x]^{\mathcal{P}_C}$. Similarly, for pairwise MACs, we can consider a sharing between two (possibly overlapping) committees $\mathcal{P}_A, \mathcal{P}_B \subset \mathcal{P}_{main}$, where $\mathcal{P}_A$ holds shares and MACs on x, while $\mathcal{P}_B$ holds the corresponding MAC keys:

$$\langle x \rangle^{\mathcal{P}_A, \mathcal{P}_B} = \Big(\{x^i, \big(M_j^i\big)_{j \in \mathcal{P}_B}\}_{i \in \mathcal{P}_A}, \{\Delta^j, (K_i^j)_{i \in \mathcal{P}_A}\}_{j \in \mathcal{P}_B} \Big)$$

When the committees are clear from context, we will sometimes omit them and simply write $\langle x \rangle$ or $[x]$.

If all the parties in $\mathcal{P}$ of size n have a sharing $\langle x \rangle^{\mathcal{P}}$, where $x = x^1 + \cdots + x^n$, any two subsets $\mathcal{P}_A, \mathcal{P}_B$ can locally convert this into a sharing $\langle x' \rangle^{\mathcal{P}_A, \mathcal{P}_B}$ of a *different* value $x' = \sum_{i \in \mathcal{P}_A} x^i$. This procedure is done by simply restricting the relevant shares and MACs to those corresponding to the two committees. We denote it as follows:

$$\mathsf{RestrictShares}(\langle x \rangle^{\mathcal{P}}, \mathcal{P}_A, \mathcal{P}_B) \to \langle x' \rangle^{\mathcal{P}_A, \mathcal{P}_B}$$

In our protocols, we rely on the fact that if the original shares of x were uniformly random, then so is the resulting value x'.

Functionality (Fig. 4.6).

The aim of $\mathcal{F}_{\mathsf{Prep}}$ is to allow arbitrary committees to obtain $[\cdot]$ and $\langle\cdot\rangle$-shared values, in the form of random authenticated field elements, and partial triples. The functionality begins with an initialization phase, which models the setting up of the necessary data to obtain up to m_R random values and m_T multiplication triples. Then, either the Rand or Trip command can be queried by a pair of dynamically-chosen committees $(\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}})$, who obtain the appropriate shares. We assume that each query uses a distinct index k, which is necessary to ensure that in our protocol, the corresponding preprocessing data is not reused when another committee produces a triple.[2]

A key difference between our functionality and previous works like SPDZ [DPSZ12, DKL+13] is that our triples are only *partially authenticated.* In a random triple (a, b, c) where $c = a \cdot b$, the values a and b are authenticated with pairwise MACs, while c is only additively shared. This is a crucial aspect which allows our protocol to support dynamically-chosen parties, and also achieving a communication overhead that is significantly less than the circuit size. One drawback of this preprocessing, compared to SPDZ, is that the size of each partial triple is $O(n)$ field elements per party, due to the pairwise MACs and products. However, once the online phase committee in which the triples will be used is known, they can be compressed to standard, constant-sized SPDZ triples.

Corrupt behaviour. As is common in SPDZ-like protocols [DPSZ12], we allow corrupted parties to choose their own randomness, i.e. shares, MACs and MAC keys, after which the honest parties' shares are picked at random to give a valid sharing. Moreover, we also allow the adversary to introduce errors into multiplication triples, by choosing error terms which are multiplied with the honest parties' shares of a and b, and then added to the result of c.

Preprocessing Protocol

Our protocol for realising $\mathcal{F}_{\mathsf{Prep}}$ consists of two main building blocks: a 2-party OLE functionality, and an n-party vector-OLE (VOLE) functionality; we elaborate on these below, and later (in Section 4.3) show how they can be realized. These are used for computing the unauthenticated shares of c in multiplication triples, and authenticated shares of random values, respectively.

Programmable OLE. We use a functionality for *random, programmable oblivious linear evaluation* (OLE), $\mathcal{F}^{\mathsf{prog}}_{\mathsf{OLE}}$, shown in Fig. 4.7. This is a two-party functionality, which computes a batch of secret-shared products, i.e. random tuples $(u_i, v_i), (w_i, x_i)$, where $w_i = u_i x_i + v_i$, over the field $\mathbb{F}_p$. The *programmability* requirement is that, for any given instance of the functionality, the party who obtains u_i or v_i can program these to be derived from a chosen random seed. This allows e.g. the same random u_i's to be used in a different instance of $\mathcal{F}^{\mathsf{prog}}_{\mathsf{OLE}}$. We model the programmability with a function $\mathsf{Expand} : S \to \mathbb{F}_p^m$, which deterministically expands the chosen seed into a vector of field elements. When instantiating the functionality, the expansion function will correspond to some kind of secure PRG.

Multi-party programmable VOLE. Vector oblivious linear evaluation (VOLE) can be seen as a batch of OLEs with the same x_i value in each tuple, that is, a vector $\boldsymbol{w} = \boldsymbol{u}x + \boldsymbol{v}$, where

[2]In our online phases, we assume the parties have a means of agreeing upon the ordering of committees to ensure that the indices queried to $\mathcal{F}_{\mathsf{Prep}}$ are not reused.

Functionality $\mathcal{F}_{\mathsf{Prep}}$

Parameters: Finite fields $\mathbb{F}_p$ and $\mathbb{F}_{p^r}$, parties $P_1, \ldots, P_n$, adversary $\mathcal{A}$ and set of honest parties $\mathcal{P}_H$.

Functionality: Generates triples with unauthenticated c, and authenticated random values.

Init: On receiving $(\mathsf{Init}, m_T, m_R)$ from P_i, for $i \in [1, n]$, where m_T is the upper bound on the number of triples and m_R on random values, sample a MAC key $\Delta^i \leftarrow_R \mathbb{F}_{p^r}$, send Δ^i to P_i and ignore subsequent Init commands from P_i.

Random Value: On input $(\mathsf{Rand}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}, k)$ from every $P_i \in \mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}$, where $k \in [1, m_R]$ and Rand has not been queried before with the same k:

1. Sample shares $r^i \leftarrow_R \mathbb{F}_p$, for $i \in \mathcal{P}_{\mathsf{curr}}$.
2. For each $i \in \mathcal{P}_{\mathsf{curr}}$ and $j \in \mathcal{P}_{\mathsf{next}} \setminus \{i\}$, sample $K_i^j \leftarrow_R \mathbb{F}_{p^r}$ and let $M_j^i = K_i^j + \Delta^j \cdot r^i \in \mathbb{F}_{p^r}$.
3. Let $\langle r \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}} = \left(r^i, (M_j^i, K_i^j)_{j \in \mathcal{P}_{\mathsf{next}} \setminus \{i\}} \right)_{i \in \mathcal{P}_{\mathsf{curr}}}$, and output the relevant shares, MACs and MAC keys to the parties in $\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}$.

Triple: On input $(\mathsf{Trip}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}, k)$, from every $P_i \in \mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}$, where $k \in [1, m_T]$ and Trip has not been queried before with the same k:

1. Run the steps from **Random Value** twice, to create sharings $\langle a \rangle, \langle b \rangle$.
2. *Additive errors:* Wait for $\mathcal{A}$ to input $\{\delta_a^i, \delta_b^i\}_{i \in \mathcal{P}_H \cap \mathcal{P}_{\mathsf{curr}}}$, each in $\mathbb{F}_p$. Let $c = a \cdot b + \sum_{i \in \mathcal{P}_H \cap \mathcal{P}_{\mathsf{curr}}} (a^i \cdot \delta_b^i + b^i \cdot \delta_a^i)$.
3. Sample shares $c^i \in \mathbb{F}_p$, for $i \in \mathcal{P}_{\mathsf{curr}}$, such that $\sum_{i \in \mathcal{P}_{\mathsf{curr}}} c^i = c$. Let $[c]^{\mathcal{P}_{\mathsf{curr}}} := (c^i)_{i \in \mathcal{P}_{\mathsf{curr}}}$.
4. Output $\langle a \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}, \langle b \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}, [c]^{\mathcal{P}_{\mathsf{curr}}}$ to the parties in $\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}$.

Corrupt parties: In addition to additive errors, corrupt parties may choose their own randomness for all sharings, namely r^i in Rand, a^i, b^i, c^i in Trip, as well as any MACs and MAC keys they receive. The honest parties' shares/MACs/keys are adjusted accordingly.

Figure 4.6: Functionality for the preprocessing

$x \in \mathbb{F}_p$ is a scalar given to one party. Here, while x lies in the field $\mathbb{F}_p$, the remaining values are in the extension field $\mathbb{F}_{p^r}$, since we use VOLE to generate MACs. In multi-party VOLE, shown as $\mathcal{F}_{\mathsf{nVOLE}}$ in Fig. 4.8, every pair of parties (P_i, P_j) is given a random VOLE instance $\boldsymbol{w}_j^i = \boldsymbol{u}^i x^j + \boldsymbol{v}_i^j$. The functionality guarantees *consistency*, in the sense that the same $\boldsymbol{u}^i$ or x^j values will be used in each of the instances involving P_i or P_j. While unlike the OLE functionality, the $\boldsymbol{u}^i, x^i$ values in $\mathcal{F}_{\mathsf{nVOLE}}$ are not programmable, we do require that the functionality outputs to P_i a short seed representing $\boldsymbol{u}^i$, so that P_i can later use this as an input to program $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$.

Protocol. Given these building blocks, we use the preprocessing protocol Π_{Prep} (Fig. 4.9) to generate partially authenticated triples and authenticated random values between dynamically

Functionality $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$

Parameters: Finite field $\mathbb{F}_{p^r}$, and expansion function $\mathsf{Expand} : S \to \mathbb{F}_p^m$ with seed space S and output length m.
The functionality runs between parties P_A and P_B.

On receiving s_a from P_A and s_b from P_B, where $s_a, s_b \in S$:

1. Compute $\boldsymbol{u} = \mathsf{Expand}(s_a)$, $\boldsymbol{x} = \mathsf{Expand}(s_b)$ and sample $\boldsymbol{v} \leftarrow_R \mathbb{F}_p^m$.
2. Output $\boldsymbol{w} = \boldsymbol{u} * \boldsymbol{x} + \boldsymbol{v}$ to P_A and $\boldsymbol{v}$ to P_B.

Corrupt parties: If P_B is corrupt, $\boldsymbol{v}$ may be chosen by $\mathcal{A}$. For a corrupt P_A, $\mathcal{A}$ can choose $\boldsymbol{w}$ (and then $\boldsymbol{v}$ is recomputed accordingly).

Figure 4.7: Functionality for programmable OLE

chosen committees. As discussed earlier, the key observation is that it suffices to generate a batch of *pairwise* secret-shared products, between every pair of parties, which can later be combined to produce preprocessing amongst an arbitrary subset of the parties.

The protocol is relatively straightforward, involving no interaction other than calling the relevant functionalities. In the Init phase of the protocol, each party P_i initializes $\mathcal{F}_{\mathsf{nVOLE}}$, obtaining a random MAC key Δ^i. Parties use the **Extend** command of $\mathcal{F}_{\mathsf{nVOLE}}$ to authenticate their shares with every other party. Towards this, each P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ twice, which picks two random seeds s_a^i, s_b^i and expands them into the shares $\boldsymbol{a}^i, \boldsymbol{b}^i$. It outputs to P_i the pairwise MACs on its shares of the triples, along with the seeds. Each pair (P_i, P_j) then use $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ to obtain 2-party sharings of the products $\boldsymbol{a}^i * \boldsymbol{b}^j$, for each $j \neq i$.

Later, when a triple is required by the committees $\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}$, every party in the committee $\mathcal{P}_{\mathsf{curr}}$ sums up its pairwise shares of the product terms corresponding to one triple, obtaining a share of $a \cdot b$, where a, b are the sum of the corresponding shares within that committee. The second committee $\mathcal{P}_{\mathsf{next}}$ does not have any shares of $a \cdot b$, but instead obtains the MAC keys on the a, b shares from the previous $\mathcal{F}_{\mathsf{nVOLE}}$ outputs. To obtain authenticated random values, a similar procedure is done using only $\mathcal{F}_{\mathsf{nVOLE}}$ to add MACs.

Note that, if a corrupt party P_i inputs an inconsistent seed s_a^i or s_b^i into $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, the resulting triple will be incorrect. This is modelled by the additive errors that may be introduced in $\mathcal{F}_{\mathsf{Prep}}$.

Below, we prove the following.

Theorem 4.1. *Suppose that $\mathsf{Expand} : S \to \mathbb{F}_p^m$ is a secure pseudorandom generator. Then, the protocol Π_{Prep} securely implements the functionality $\mathcal{F}_{\mathsf{Prep}}$ in the $(\mathcal{F}_{\mathsf{nVOLE}}, \mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}})$-hybrid model, when up to $n-1$ out of n parties are corrupted.*

Proof. Since the protocol involves no interaction other than with $\mathcal{F}_{\mathsf{nVOLE}}$ and $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, simulation is quite straightforward. Let A be the set of corrupt parties. We construct a simulator, $\mathcal{S}$, as follows. For each $i \in A$, $\mathcal{S}$ receives Δ^i from $\mathcal{A}$ and forwards it to $\mathcal{F}_{\mathsf{Prep}}$. We focus on the setup for triple generation; the simulation for random values is simpler. $\mathcal{S}$ receives the corrupt parties' seeds s_a^i, s_b^i as input to $\mathcal{F}_{\mathsf{nVOLE}}$, as well as the MACs and MAC key outputs which are chosen by the corrupt parties. $\mathcal{S}$ then computes the expanded shares $\boldsymbol{a}^i = \mathsf{Expand}(s_a^i)$ and $\boldsymbol{b}^i = \mathsf{Expand}(s_b^i)$. For each $i \in A$ and honest P_j, it receives seeds $s_a^{i,j}, s_b^{i,j}$ as input to the $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$

Functionality $\mathcal{F}_{\mathsf{nVOLE}}$

Parameters: Finite field $\mathbb{F}_{p^r}$, and expansion function $\mathsf{Expand} : S \to \mathbb{F}_p^m$ with seed space S and output length m. The functionality runs between $P_1, \ldots, P_n$.

Initialise: On receiving Init from P_i, for $i \in [1, n]$, sample $\Delta^i \leftarrow_R \mathbb{F}_{p^r}$, send it to P_i, and ignore all subsequent Init commands from P_i.
Extend: On receiving (Extend) from every $P_i \in \mathcal{P}$:

1. Sample $\mathsf{seed}^i \leftarrow_R S$, for each $P_i \in \mathcal{P}$.
2. Compute $\boldsymbol{u}^i = \mathsf{Expand}(\mathsf{seed}^i)$.
3. Sample $(\boldsymbol{v}_i^j)_{j \neq i} \leftarrow_R \mathbb{F}_{p^r}^m$ for $i \in \mathcal{P}, j \neq i$. Retrieve Δ^j and compute $\boldsymbol{w}_j^i = \boldsymbol{u}^i \cdot \Delta^j + \boldsymbol{v}_i^j$.
4. If P_j is corrupt, receive a set I from $\mathcal{A}$. If $\mathsf{seed} \in I$, send $\mathsf{success}$ to P_j and continue. Else, send abort to both parties, output seed to P_j and abort.
5. Output $\left((\mathsf{seed}^i, \boldsymbol{w}_j^i), \boldsymbol{v}_j^i\right)_{j \neq i}$ to P_i, for $P_i \in \mathcal{P}$.

Corrupt parties: A corrupt P_i can choose Δ^i and seed^i. It can also choose $\boldsymbol{w}_j^i$ (and $\boldsymbol{v}_i^j$ is recomputed accordingly) and $\boldsymbol{v}_j^i$.
Global key query: If P_i is corrupted, receive $(\mathsf{guess}, \boldsymbol{\Delta}')$ from $\mathcal{A}$ with $\boldsymbol{\Delta}' \in \mathbb{F}_{p^r}^n$. If $\boldsymbol{\Delta}' = \boldsymbol{\Delta}$, where $\boldsymbol{\Delta} = (\Delta^1, \ldots, \Delta^n)$, send $\mathsf{success}$ to P_i and ignore any subsequent global key query. Else, send $(\mathsf{abort}, \boldsymbol{\Delta})$ to P_i, abort to P_j and abort.

Figure 4.8: Functionality for n-party VOLE

instances between P_i and P_j. For any instance where $s_a^{i,j} \neq s_a^i$, $\mathcal{S}$ computes the additive error multipliers $\boldsymbol{\delta}_b^{i,j} = \mathsf{Expand}(s_a^{i,j}) - \boldsymbol{a}^i$, and similarly computes $\boldsymbol{\delta}_a^{i,j} = \mathsf{Expand}(s_b^{i,j}) - \boldsymbol{b}^i$. For $j \in [n] \setminus A$, let $\boldsymbol{\delta}_b^j = \sum_{i \in A} \boldsymbol{\delta}_b^{i,j}$, and $\boldsymbol{\delta}_a^j = \sum_{i \in A} \boldsymbol{\delta}_a^{i,j}$.

Finally, $\mathcal{S}$ sends the error terms $\boldsymbol{\delta}_a^j, \boldsymbol{\delta}_b^j$ to $\mathcal{F}_{\mathsf{Prep}}$, as well as the corrupted parties' expanded shares $\boldsymbol{a}^i, \boldsymbol{b}^i$ (for $i \in A$), MACs, MAC keys and c^i shares (all computed the same way as in the protocol).

We now argue indistinguishability of the ideal and real executions. Since the corrupt parties receive no information during the protocol, we only need to look at the distribution of the parties' outputs. Let $\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}$ be two committees which query the **Triples** command, and suppose each committee has at least one honest party (for an entirely corrupt committee, indistinguishability of the corresponding outputs is trivial). Each sharing $\langle a \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}, \langle b \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}$ is defined from a subset of the original sharings $\langle a \rangle, \langle b \rangle$, where each honest party's share a^i, b^i was derived as an output of Expand on an independent random seed. Hence, by a standard hybrid argument, these shares are computationally indistinguishable from random values. The MACs and MAC keys held by the two committees on $\langle a \rangle, \langle b \rangle$ are perfectly indistinguishable, because in both worlds, corrupt parties choose their own values, while values between a pair of honest parties are sampled at random. Finally, we need to consider the shares c^i, for $i \in \mathcal{P}_{\mathsf{curr}}$. In the real world, we have

Protocol Π_{Prep}

Parameters: Finite field $\mathbb{F}_{p^r}$, number of triples m_T, random values m_R, and expansion function $\mathsf{Expand} : S \to \mathbb{F}_p^m$ with seed space S and output length m.
Init: Run the following two stages among all the parties in $\mathcal{P}_{\mathsf{main}}$.
Triples setup: repeat the following, until $\geq m_T$ outputs have been obtained (each iteration produces m).

1. Each P_i calls $\mathcal{F}_{\mathsf{nVOLE}}$ with Init, receiving Δ^i.
2. Each P_i, for $i \in [1, n]$, calls $\mathcal{F}_{\mathsf{nVOLE}}$ twice, with input Extend and receives the seeds s_a^i, s_b^i. Use the outputs to define vectors of shares $\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle$ such that $\boldsymbol{a}^i = \mathsf{Expand}(s_a^i)$ and $\boldsymbol{b}^i = \mathsf{Expand}(s_b^i)$.
3. Every ordered pair (P_i, P_j) for $i, j \in [1, n]$ calls $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ with P_i sending s_a^i and P_j sending s_b^j, and it sends back $\boldsymbol{u}^{i,j}$ to P_i and $\boldsymbol{v}^{j,i}$ to P_j, such that $\boldsymbol{u}^{i,j} + \boldsymbol{v}^{j,i} = \boldsymbol{a}^i * \boldsymbol{b}^j$.

Random values setup: repeat the following, until $\geq m_R$ outputs have been obtained.

1. Every P_i, for $i \in [1, n]$, samples a seed $s_r^i \in S$ and calls $\mathcal{F}_{\mathsf{nVOLE}}$ with input (Extend, s_r^i) from P_i, forming $\langle \boldsymbol{r} \rangle$.

Triples: To get the k-th triple in committees $\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}$:

1. Let $\langle a' \rangle, \langle b' \rangle$ be the k-th shares from $\langle \boldsymbol{a} \rangle, \langle \boldsymbol{b} \rangle$. The parties run $\mathsf{RestrictShares}(\langle a' \rangle, \langle b' \rangle, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}})$ to obtain $\langle a \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}, \langle b \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}$.
2. Each $P_i \in \mathcal{P}_{\mathsf{curr}}$ computes $c^i = a^i \cdot b^i + \sum_{j \in \mathcal{P}_{\mathsf{curr}} \setminus \{i\}} (\boldsymbol{u}^{i,j}[k] + \boldsymbol{v}^{i,j}[k])$.
3. The parties output the triple $(\langle a \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}, \langle b \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}, [c]^{\mathcal{P}_{\mathsf{curr}}})$.

Random Values: To get the k-th random value in committees $\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}$, the parties take $\langle r' \rangle$, the k-th random value from $\langle \boldsymbol{r} \rangle$, and run $\mathsf{RestrictShares}$ to convert this to $\langle r \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{next}}}$.

Figure 4.9: Protocol for preprocessing

$$\begin{aligned}
c = \sum_{i \in \mathcal{P}_{\mathsf{curr}}} c^i &= \sum_{i \in \mathcal{P}_{\mathsf{curr}}} (a^i b^i + \sum_{j \neq i} (u^{i,j} + v^{i,j})) \\
&= \sum_{i \in \mathcal{P}_{\mathsf{curr}}} (a^i b^i + \sum_{j \neq i} (u^{i,j} + v^{j,i})) \\
&= \sum_{i \in \mathcal{P}_{\mathsf{curr}}} (a^i b^i + \sum_{j \neq i} (a^{i,j} b^{j,i})
\end{aligned}$$

where $a^{i,j}, b^{i,j}$ equal a^i, b^i if P_i is honest, or if P_i is corrupt, derived from the seed used by P_i with P_j in $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$. Plugging in $a^{i,j} = \delta_b^{i,j} + a^i$ and $b^{j,i} = \delta_a^{j,i} + b^j$, we have

$$\begin{aligned}
c &= \sum_{i \in \mathcal{P}_{\mathsf{curr}}} (a^i b^i + \sum_{j \neq i} (a^i + \delta_b^{i,j}) \cdot (b^j + \delta_a^{j,i}) \\
&= ab + \sum_{i \in \mathcal{P}_{\mathsf{curr}}} \sum_{j \neq i} (a^i \delta_a^{j,i} + b^j \delta_b^{i,j}) \\
&= ab + \sum_{i \in \mathcal{P}_{\mathsf{curr}}} (a^i \delta_a^i + b^i \delta_b^i)
\end{aligned}$$

where δ_a^i, δ_b^i are defined as in the error vectors from the simulation, and we have assumed that, for any i, j where both P_i and P_j are corrupt, $\delta_a^{i,j}$ and $\delta_b^{j,i}$ are both zero (since here, simulation is trivial).

It follows that the way c is computed in the real world, above, is identical to that in the ideal world. Furthermore, the randomness of the individual c^i shares is guaranteed, because of the randomly sampled outputs of $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$ between two honest parties. □

Instantiating Multi-Party VOLE

In multi-party VOLE, each party P_i runs an instance of random VOLE with every other party P_j. We model two-party random VOLE as the functionality $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ [RS21], and show how to realize it in Section 4.3. To allow parties to use the *same* random input in different VOLE instances, the functionality is also programmable, similarly to $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$.

The main challenge in realizing $\mathcal{F}_{\mathsf{nVOLE}}$ is to guarantee that each party uses the same programmed input across every instance of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ with other parties. For instance, a corrupt party P_i could potentially use different Δ^i values as the sender, or different seeds for $\boldsymbol{u}^i$ as the receiver across instances. To prevent this, we use a consistency check to prevent parties from using different inputs across the instances. The check involves taking a random linear combination of the outputs of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ and opening the sum, and is similar to the $\Pi_{\mathsf{TripleBucketing}}$ protocol from [HSS17], except we work over a general finite field rather than $\mathbb{F}_2$.

Another difference is that we formalize the resulting protocol and show it realizes the multi-party VOLE functionality, while in [HSS17], the check was only used as part of a larger protocol. To prove this, we had to introduce the **Global key query** command in $\mathcal{F}_{\mathsf{nVOLE}}$, which allows corrupt parties to try to guess the honest parties' global scalars (MAC keys).

The final protocol for Π_{nVOLE} appears in Fig. 4.10.

Consistency Check:

Since $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ does not guarantee that each party uses the same seed s^i or scalar Δ^i with every other party, we need some sort of a consistency check to detect malicious behaviour. The high level idea is for parties to compute random linear combinations on the outputs of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, securely open the sum and check that it is zero. This check is similar to the idea from [HSS17], wherein it was used to check TinyOT triples.

The protocol starts with each (P_i, P_j) running $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ between them twice, once with P_i as the sender and once as the receiver. Recall that for a value v, P_i holds the share $\langle v \rangle = (v^i, \{M_j^i, K_j^i\}_{j \neq i})$. Using the outputs of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, each P_i can define its shares of $\langle r_1 \rangle, \ldots, \langle r_m \rangle, \langle t \rangle \in \mathbb{F}_{p^r}$ locally. To compute a random linear combination, parties call $\mathcal{F}_{\mathsf{Rand}}$ and receive $\chi_1, \ldots, \chi_m \in \mathbb{F}_{p^r}$. They can locally compute shares of $\langle C \rangle$, and reconstruct C by broadcasting the shares. We wish to check $\sum_{i=1}^n Z_j^i = 0$ for $j \in [1, n]$, where $\{Z_j^i\}_{i \neq j} = M_j^i$ and $Z_i^i = (C^i - C) \cdot \Delta^i - \sum_{j \neq i} K_j^i$. Parties

Protocol Π_{nVOLE}

Parameters: Extension field $\mathbb{F}_{p^r}$, parties $P_1, \ldots, P_n$.
Initialise: Each party P_i samples $\Delta^i \leftarrow_R \mathbb{F}_{p^r}$. Every ordered pair of parties (P_i, P_j) calls $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ with $(\mathsf{Init}, \Delta^i), \mathsf{Init}$ respectively.
Random Values: To create m authenticated random values $\langle r_1 \rangle, \ldots, \langle r_m \rangle$,

1. Each party P_i samples a seed s^i.
2. Each ordered pair of parties (P_i, P_j) call $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, with P_i sending (Extend, s^i) and P_j sending Extend. P_i receives $\{r_k^i, M_j^{i,k}\}$ and P_j receives $K_i^{j,k}$ for $k \in [1, m+1]$.
3. The outputs of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ define sharings $\langle r_1 \rangle, \ldots, \langle r_m \rangle, \langle t \rangle \in \mathbb{F}_{p^r}$, where each $r_j = \sum_{i=1}^n r_j^i$ and $t = \sum_{i=1}^n r_{m+1}^i$.
4. Each P_i does the following to check the consistency of inputs to $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$:
   a) Call $\mathcal{F}_{\mathsf{Rand}}$ together with other parties to get random values $\chi_1, \ldots, \chi_m \in \mathbb{F}_{p^r}$.
   b) Locally compute
   $$\langle C \rangle = \sum_{i=1}^m \chi_i \cdot \langle r_i \rangle + \langle t \rangle$$
   c) P_i has a share C^i, the MACs and keys $(M_j^i, K_j^i)_{j \neq i}$ from $\langle C \rangle$.
   d) P_i rerandomizes the share locally by sending a zero share to the other parties. Call the randomised shares $\hat{C}^i$.
   e) Broadcasts $\hat{C}^i$ and reconstructs $C = \sum_{i=1}^n \hat{C}^i$
   f) P_i calls $\mathcal{F}_{\mathsf{Commit}}$ with $n+1$ values:
   $$C^i, \quad (Z_j^i)_{j \neq i} = M_j^i, \quad Z_i^i = (C^i - C) \cdot \Delta^i - \sum_{j \neq i} K_j^i$$
5. Parties open their commitments and check that $\sum_{i=1}^n Z_j^i = 0$, for $j \in [1, n]$. In addition, each P_i checks that $Z_i^j = K_j^i + C^j \cdot \Delta^i$. If any of the checks fail, abort.

Figure 4.10: Protocol for Consistent VOLE

commit and open their shares, and locally check that each $\sum_{i=1}^n Z_j^i = 0$. If any of them fail, they abort.

The following is an analysis of the consistency check followed by a proof.

Analysis of the Consistency Check

Since $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ does not guarantee that each party P_i uses the same seed s^i with every other party, we need some sort of a consistency check to detect malicious behaviour. The high level idea is for parties to compute a random linear combination on the outputs of $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$, securely open the sum and check that it is zero. The check is similar to the one from [HSS17], wherein it was used to check TinyOT triples.

Recalling the notation for a 2-party MAC between (P_i, P_j), P_i holds the values (x^i, M^i_j), where $M^i_j(x^i) = K^j_i(x^i) + x^i \cdot \Delta^j$. K^j_i is the local key that P_j has with P_i, and Δ^j is the global key that is supposed to be kept the same across interactions with different parties.

We formalise the security of the consistency check used in Fig. 4.10. There are two sources of errors a corrupt P_B can use, which are:

1. Providing inconsistent inputs (Δ) when acting as the sender in the Initialise command of $\mathcal{F}^{\mathsf{prog}}_{\mathsf{VOLE}}$ with 2 different honest parties.
2. Providing inconsistent values (s) when acting as the receiver in the Extend command of $\mathcal{F}^{\mathsf{prog}}_{\mathsf{VOLE}}$ with 2 different honest parties.

In both instances, we are only concerned with the cases in which a dishonest party interacts with an honest one. If both parties are corrupt, $\mathcal{F}^{\mathsf{prog}}_{\mathsf{VOLE}}$ need not be simulated in the proof.

Difference between [HSS17] and this: In [HSS17], the adversary can use different values as inputs when acting as the receiver with different honest parties. This translates to a chosen additive error by the adversary. However, in our case the adversary inputs a seed s, from which the value $\boldsymbol{u}$ is computed as $\mathsf{Expand}(s)$. Therefore, this will not be an arbitrarily chosen additive error but limited to a subset of values over the field.

For the analysis, we continue to treat this error as an arbitrarily chosen additive error.

These attacks are modelled by defining the inputs used by a corrupt P_j, with every honest party. Let P_{i_0} be the party for which P_j uses the inputs s^{j,i_0}, and Δ^{j,i_0}, which we consider to be the *actual* inputs. As a result of using a different s with different parties, the values $\boldsymbol{r}, t$ will be different. Let the values used by P_j with P_{i_0} be r^{j,i_0}_l, t^{j,i_0} $\forall l \in [m]$. For simplicity, we omit the i_0 in the superscript for these values. Ideally P_j should use the same inputs with every other honest party. We can model the errors as:

$$\begin{aligned}
\varepsilon^{j,i_0} &= 0, \quad \varepsilon^{j,i} = \Delta^{j,i} - \Delta^j, \quad i \notin (\mathcal{A} \cup i_0) \\
\delta^{j,i_0} &= 0, \quad \delta^{j,i}_l = r^{j,i}_l - r^j_l, \quad l \in [m], i \notin (\mathcal{A} \cup i_0) \\
\hat{\delta}^{j,i_0} &= 0, \quad \hat{\delta}^{j,i}_l = t^{j,i} - t^j, \quad i \notin (\mathcal{A} \cup i_0)
\end{aligned}$$

Where $\varepsilon^{j,i}$ is the error in the global key used by P_j with P_i. This error is fixed in the Initialise command, whereas the error δ can be different in every instance of Extend. If P_i, P_j are both corrupt, or both honest, the errors are set to 0. Therefore, the outputs of $\mathcal{F}^{\mathsf{prog}}_{\mathsf{VOLE}}$ between (P_j, P_i) satisfy:

$$M^J_i(r^{j,i}_l) = K^i_j(r^{j,i}_l) + r^{j,i}_l \cdot \Delta^{i,j}$$

or equivalently,

$$M^j_i(r^j_l + \varepsilon^{j,i}_l) = K^i_j(r^j_l + \varepsilon^{j,i}_l) + (r^j_l + \delta^{j,i}_l) \cdot (\Delta^j + \varepsilon^{i,j})$$

$\delta^{j,i} \neq 0$ if P_j (the receiver) cheated, and $\varepsilon^{i,j} \neq 0$ if P_i (the sender) cheated.

The first case is of a corrupt sender P_j, which uses inconsistent global keys $\Delta^{j,i}$ when acting as a sender with different honest parties P_i, $i \notin (\mathcal{A} \cup i_0)$. The inconsistency is proved impossible via:

Lemma 4.1. *If* Π_{nVOLE} *succeeds, then all the global keys* $\Delta^{j,i}$ *are consistent and well defined, i.e* $\varepsilon^{j,i} = 0$ *for every* $i, j \in [1, n]$.

Proof. We start by analysing possible deviations by $P_j \in \mathcal{A}$ in Step 4g in Fig. 4.10, where we want to catch inconsistent $\Delta^{j,i}$ used with different honest parties.

In Step 4e, parties broadcast their shares of C, and the corrupted parties can send the wrong shares so that $\sum_{j=1}^{n} \hat{C}^j = C + e$, where e is the additive error from P_j. Another thing the corrupted parties can do is cheat in the commitments, by committing to $\hat{Z}_j^l$ values such that $\sum_{l \in \mathcal{A}} \hat{Z}_j^l = \sum_{l \in \mathcal{A}} Z_j^l + E^j$.

Therefore, the check now becomes:

$$\begin{aligned}
0 &= \sum_{i=1}^{n} \hat{Z}_j^i \\
&= E^j + Z_j^j + \sum_{i \neq j} Z_j^i \\
&= E^j + \left[(C^j - C - e) \cdot \Delta^j - \sum_{i \neq j} K^j(C^i) \right] + \sum_{i \neq j} M_j^i(C^i) \\
&= E^j + (C^j - C - e) \cdot \Delta^j + \sum_{i \neq j} (M_j^i(C^i) - K_i^j(C^i)) \\
&= E^j + (C^j - C - e) \cdot \Delta^j + \sum_{i \neq j} C^i \cdot \Delta^{j,i} \\
&= E^j + (C^j + \sum_{i \neq j} C^i - C - e) \cdot \Delta^j + \sum_{i \neq j} C^i \cdot \varepsilon^{j,i} \\
&= E^j - e \cdot \Delta^j + \sum_{i \neq j} C^i \cdot \varepsilon^{j,i}
\end{aligned}$$

□

where $\varepsilon^{j,i}$ indicates the error as compared to the Δ^j used in computing Z_j^j. Using inconsistent global keys means that $\exists i' \notin (\mathcal{A} \cup i_0), \varepsilon^{j,i'} \neq 0$. Therefore the attack would require $e \cdot \Delta^j - E^j = C^{i'} \cdot \varepsilon^{j,i'}$. P_j does not know anything about the shares of C at the time of committing to $\hat{Z}_j^l$ due to using the re-randomised shares of C for reconstruction in step 4e. Therefore, the probability that the check passes with the errors is $1/\mathbb{F}$ as the adversary will have to guess the share of C.

The second case is proving that P_j as a corrupted receiver cannot input inconsistent values $\boldsymbol{e}^{j,i}$ to different honest parties.

Lemma 4.2. *If* Π_{nVOLE} *succeeds, every ordered pair* (P_i, P_j) *holds a secret sharing of* $r_l^j \cdot \Delta^i$ *for every* $l \in [1, m]$. *In other words,* $\delta_l^{j,i} = 0$ *for every* i, j, l.

Proof. We can define the MAC on C^j held by P_j with party P_i as,

$$M_i^j(C^j) = \sum_{l=1}^{m} \chi_l \cdot M_i^j(r_l^{j,i}) + M_i^j(t^{j,i})$$

and the key held by P_i as,

$$K_j^i(C^j) = \sum_{l=1}^{m} \chi_l \cdot K_j^i(r_l^{j,i}) + K_j^i(t^{j,i})$$

In step 4f of Π_{nVOLE}, a corrupted P_j can commit to incorrect MACs $\hat{Z}_i^j(C^j) = M_i^j(C^j) + E_i^j$ and $\hat{C}^j = C^j + e^j$. In order to succeed, the check $\hat{Z}_i^j = K_j^i(C^j) + \hat{C}^j \cdot \Delta^i$ from step 4g must hold. This implies,

$$\begin{aligned}
&M_i^j(C^j) + E_i^j = K_j^i(C^j) + (C^j + e^j) \cdot \Delta^i \\
&\implies E_i^j - (C^j + e^j) \cdot \Delta^i = K_j^i(C^j) - M_i^j(C^j) = -\left(\sum_{l=1}^{m} \chi_l \cdot r_l^{j,i} + t^{j,i}\right) \cdot \Delta^i \\
&\implies E_i^j = \left(C^j + e^j - \sum_{l=1}^{m} \chi_l \cdot (r_l^j + \delta_l^{j,i}) + (t^{j,i} + \hat{\delta}^{j,i})\right) \cdot \Delta^i = (e^j - \sum_{l=1}^{m} \chi_l \cdot \delta_l^{j,i} + \hat{\delta}^{j,i}) \cdot \Delta^i
\end{aligned}$$

A malicious P_j has two options to cheat, both with probability of $1/\mathbb{F}$ to succeed:

1. Setting $E_i^j = (e^j - \sum_{l=1}^{m} \chi_l \cdot \delta_l^{j,i} + \hat{\delta}^{j,i}) \cdot \Delta^i \neq 0$, which requires guessing Δ^i, known only to P_i.

2. Set $E_i^j = 0$ and $e^j = \sum_{l=1}^{m} \chi_l \cdot \delta_l^{j,i} + \hat{\delta}^{j,i}$ for every $i \notin \mathcal{A}$. Since $\delta_l^{j,i_0} = \hat{\delta}^{j,i_0} = 0$, e^j should also be 0. Therefore, for $i \notin (\mathcal{A} \cup i_0)$ it should hold that,

$$0 = \sum_{l=1}^{m} \chi_l \cdot \delta_l^{j,i} + \hat{\delta}^{j,i} = \hat{\delta}^{j,i} = -\sum_{l=1}^{m} \delta_l^{j,i} \cdot \chi_l \in \mathbb{F}_{p^r}$$

Since $\boldsymbol{\chi}$ are uniformly random values from a field, the probability that this holds is $1/\mathbb{F}$. □

Theorem 4.2. *Protocol* Π_{nVOLE} *UC-securely computes* $\mathcal{F}_{\mathsf{nVOLE}}$ *in the presence of a static malicious party corruption up to* $n-1$ *in the* $(\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{Commit}})$*-hybrid model.*

Proof. We construct a PPT Simulator ($\mathcal{S}$) that run the adversary ($\mathcal{A}$) as a subroutine, and is given access to $\mathcal{F}_{\mathsf{nVOLE}}$. It internally emulates the functionalities $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{Commit}}$ and we implicitly assume that it passes all communication between $\mathcal{A}$ and the environment ($\mathcal{Z}$).

The parties controlled by the $\mathcal{A}$ are indicated by $\mathcal{P}_{\mathcal{A}}$ and the honest parties by $\mathcal{P}_{\mathcal{H}}$. The simulator uses a flag which is set to 1 in case $\mathcal{A}$ is caught cheating before the consistency check happens, and the simulation is carried on. The simulation proceeds as follows:

Malicious $\mathcal{P}_{\mathcal{A}}$:

Init: $\mathcal{S}$ receives a vector $\boldsymbol{\Delta}^i$ for every $i \in \mathcal{A}$, which are its inputs to $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$. $\mathcal{S}$ chooses the first one in each of these vectors and forwards them to $\mathcal{F}_{\mathsf{nVOLE}}$ with the Init command. If any of these vectors are not of the form $(\Delta^i, \dots, \Delta^i)$, set the $\mathsf{flag} = 1$.

Random Values:

1. When $\mathcal{A}$ acts as receiver in step 2, $\mathcal{S}$ receives a vector $\boldsymbol{e}_j^i$ from every $P_i \in \mathcal{P}_{\mathcal{A}}$ and $j \in [1, n]$. It picks the first vector and forwards it to $\mathcal{F}_{\mathsf{nVOLE}}$ with the Extend command. If any of the vectors received from a P_i are inconsistent, set $\mathsf{flag} = 1$.

2. For $P_i \in \mathcal{P}_{\mathcal{A}}$ and $j \in [1, n]$, $\mathcal{S}$ records $\boldsymbol{w}_j^i$ when $\mathcal{A}$ acts as the receiver in step 2, and $\boldsymbol{v}_j^i$ when it acts as the sender.

3. Emulate the call to $\mathcal{F}_{\mathsf{Rand}}$ by sampling $\chi_1, \dots, \chi_m$ and sending them to $\mathcal{A}$.

4. Receive zero-shares from $\mathcal{A}$ and record them. Sample a zero-share for $P_j \in \mathcal{P}_H$ and send them to $\mathcal{A}$.

5. Sample a random share of C for each honest party and send them to $\mathcal{A}$. Receive $\hat{C}^i$ for $P_i \in \mathcal{P}_\mathcal{A}$, reconstruct $C = \sum_{i=1}^{n} \hat{C}^i$.

6. Emulate $\mathcal{F}_{\mathsf{Commit}}$ by recording $\tilde{C}^i, (Z_j^{i'})_{j \neq i}, Z_i^{i'}$ from $P_i \in \mathcal{P}_\mathcal{A}$. $\mathcal{S}$ computes C^i as it knows $\boldsymbol{\chi}$, and shares of $\mathcal{A}$ for $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$. Using those, it sets $\langle C \rangle = \sum_{i=1}^{m} \chi_i \cdot \langle r_i \rangle + \langle t \rangle$ for all parties in $\mathcal{P}_\mathcal{A}$.

7. If $\tilde{C}^i = C^i$ and $\mathsf{flag} = 0$: for each sum, $\sum_{i=1}^{n} Z_j^i$, where $j \in [1, n]$, sample shares for the honest parties as follows: sample uniformly random values for all but one honest party, and pick the last share such that the sum is zero.

8. If $\tilde{C}^i = C^i$ and $\mathsf{flag} = 1$: sample random values for $\mathcal{P}_H$ for shares of Z and send them to $\mathcal{A}$, send abort to $\mathcal{F}_{\mathsf{nVOLE}}$ and abort.

9. If $\tilde{C}^i \neq C^i$, compute $\tilde{Z}_j^i - Z_j^i$, where Z_j^i is the value computed by $\mathcal{S}$ using C^i, for all $j \in \mathcal{P}_H$. For $\mathcal{A}$ to pass the check, it must have guessed the correct Δ^j for every honest P_j.
   a) Therefore, $\mathcal{S}$ can extract $\mathcal{A}$'s guess as $\tilde{\Delta}^j = (\tilde{Z}_j^i - Z_j^i)/(\tilde{C}^i - C^i)$. Set $\tilde{\boldsymbol{\Delta}} = (\tilde{\Delta}^j, \ldots)$.
   b) Forward $(\mathsf{guess}, \tilde{\boldsymbol{\Delta}})$ to $\mathcal{F}_{\mathsf{nVOLE}}$. If $\mathcal{F}_{\mathsf{nVOLE}}$ returns $\mathsf{success}$, send true to $\mathcal{A}$, forward $\boldsymbol{w}$ to $\mathcal{F}_{\mathsf{nVOLE}}$. Compute shares of $\mathcal{P}_H$ such that $\sum_{i=1}^{n} Z_j^i = 0$ for $j \in [1, n]$ and send them to $\mathcal{A}$. Output whatever $\mathcal{A}$ outputs.
   c) Else, receive $(\mathsf{abort}, \boldsymbol{\Delta})$, where $\boldsymbol{\Delta}$ is the vector of Δ values used by $\mathcal{P}_H$. Compute shares of $\mathcal{P}_H$ using $\boldsymbol{\Delta}$, send them to $\mathcal{A}$, and abort.

10. Whenever $\mathcal{A}$ queries $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ with a set I, forward it to $\mathcal{F}_{\mathsf{nVOLE}}$.

□

The Missing Pieces: Programmable OLE and VOLE.

We now describe how to realize the two missing building blocks used in our preprocessing protocol, namely 2-party programmable OLE and VOLE.

Realizing $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$. This can be realized in a number of ways, for instance, based on linearly homomorphic encryption [BDOZ11]. However, this would give a protocol with communication that scales *linearly* in m, the number of OLEs. Instead, we rely on the recent work of [BCG+20], which uses a variant of the ring-LPN assumption to obtain communication that is *logarithmic* in m. While the OLE functionality from [BCG+20] is not programmable, we observe that their protocol easily supports programmable inputs, so suffices for our application.

Realizing $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$. Unlike the OLE protocol from [BCG+20], this work starts with a building block called *single-point* VOLE, where the vector $\boldsymbol{u}$ contains a single, non-zero element, which is assumed to be sampled at random. When we need programmability, however, we cannot assume this. We therefore modify the underlying single-point VOLE from [WYKW21b] to support programmable inputs, and show that the resulting protocol is still secure. We show how this can then be used to build programmable VOLE, with essentially the same steps as [WYKW21a].

We start with a standard random VOLE functionality called Base VOLE, as shown in Fig. 4.11. This can be realised by any of the existing protocols for VOLE [ADI+17, BCGI18, BCG+19a, WYKW21b]. Using this, we build a single-point subfield VOLE (spsVOLE), where the input of

Functionality $\mathcal{F}_{\mathsf{sVOLE}}$

Parameters: An extension field $\mathbb{F}_{p^r}$, length m, and party identifiers P_A, P_B.
Initialise: On receiving Init from P_A, and (Init, Δ) from P_B, store the global key Δ, and ignore all subsequent Init commands.
Extend: This procedure can be run multiple times. On receiving (Extend, l) from P_A, P_B, do:

1. If P_B is honest, sample $\mathsf{K}[x] \leftarrow_R \mathbb{F}_{p^r}^l$. Else, receive $\mathsf{K}[x] \in \mathbb{F}_{p^r}^l$ from $\mathcal{A}$.
2. If P_A is honest, sample $x \leftarrow_R \mathbb{F}_p^l$ and compute $\mathsf{M}[x] = \mathsf{K}[x] + \Delta \cdot x \in \mathbb{F}_{p^r}^l$. Else, receive $x \in \mathbb{F}_p^l$ and $\mathsf{M}[x] \in \mathbb{F}_{p^r}^l$ from $\mathcal{A}$ and recompute $\mathsf{K}[x] = \mathsf{M}[x] - \Delta \cdot x \in \mathbb{F}_{p^r}^l$.
3. Send $(x, \mathsf{M}[x])$ to P_A and $\mathsf{K}[x]$ to P_B.

Global key query: If P_A is corrupted, receive $(\mathsf{guess}, \Delta')$ from $\mathcal{A}$ with $\Delta' \in \mathbb{F}_{p^r}$. If $\Delta' = \Delta$, send $\mathsf{success}$ to P_A and ignore any subsequent global key query. Else, send abort to both parties and abort.

Figure 4.11: Functionality for a subfield VOLE (Base VOLE)

the receiver is a $\boldsymbol{u}$ such that $\boldsymbol{u}[\alpha] = \beta$ and is 0 everywhere else. Wolverine [WYKW21b] has a construction for random spsVOLE, where the sender's global key Δ and the receiver's input $\boldsymbol{u}$ are randomly picked. Since, in our setting, we want parties to be able to influence the randomness used to derive their inputs, we give a modified version of this protocol that supports chosen-inputs, in Fig. 4.13.

To reflect the chosen-input protocol, we need to slightly modify the spsVOLE functionality, $\mathcal{F}_{\mathsf{spsVOLE}}^{\mathsf{ci}}$, in Fig. 4.12. First, we let the party P_A choose α and β, which determine the special point in the vector $\boldsymbol{u}$, which is nonzero only at $\boldsymbol{u}[\alpha] = \beta$. The second tweak is to make $\mathcal{F}_{\mathsf{spsVOLE}}^{\mathsf{ci}}$ reveal the secret index α used by an honest P_A, in case of an abort. Previously, this was not needed, since α was always sampled at random and not a private input.

The protocol $\Pi_{\mathsf{spsVOLE}}^{\mathsf{ci}}$ uses $\mathcal{F}_{\mathsf{OT}}$, a standard OT functionality, and $\mathcal{F}_{\mathsf{EQ}}$, a functionality to check equality, which reveals an honest P_A's input to P_B. The protocol can be split into two parts, with the first part being a semi-honest VOLE protocol, and the second part involving a consistency check.

Parties P_A, P_B start by generating $\langle \beta \rangle \in \mathbb{F}_p$, where β is an input of P_A. Doing so is straightforward and involves one call to $\mathcal{F}_{\mathsf{sVOLE}}$. P_A then defines the single-point vector $\boldsymbol{u} \in \mathbb{F}_p^m$ such that $\boldsymbol{u}[\alpha] = \beta$, where $\alpha \in [0, n)$ is also its input. Next we need P_B to generate $\boldsymbol{v} \in \mathbb{F}_p^m$ in such a way that P_A learns all $\boldsymbol{v}[i]$ values except $\boldsymbol{v}[\alpha]$. Towards this, parties run the GGM subroutine, starting with P_B sampling $s \leftarrow_R \{0,1\}^\kappa$ and computing all the nodes in the GGM tree of depth h with s as the root node. The j-th node in the tree at the i-th level is denoted by s_j^i. P_B defines $s_0^0 = s$ as the root, and computes $\left(s_{2j}^i, s_{2j+1}^i\right) = G(s_j^{i-1})$, for $i \in [1, h)$, $j \in [0, 2^{j-1})$, where $G : \{0,1\}^\kappa \to \{0,1\}^{2k}$ is a PRG. The leaf nodes are computed as $(\boldsymbol{v}[2j], \boldsymbol{v}[2j+1]) = G'(s_j^{h-1})$ for $j \in [0, 2^{h-1})$, where $G' : \{0,1\}^\kappa \to \mathbb{F}_{p^r}^2$ is a PRG. The $\mathsf{GGM}(1^n, s)$ output can be written as, $\left(\{v_j\}_{j \in [0,n)}, \{(K_0^i, K_1^i)\}_{i \in [h]}\right)$, where (K_0^i, K_1^i) are the XOR of the values at the even and odd nodes at level i respectively. For the leaf nodes, instead of XOR, addition over $\mathbb{F}_{p^r}$ is computed. Then, parties run h instances of $\mathcal{F}_{\mathsf{OT}}$ with P_A sending $\bar{\alpha}^i$ for $i \in [1, h]$ and P_B sending (K_0^i, K_1^i)

Functionality $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$

Parameters: An extension field $\mathbb{F}_{p^r}$, length m, and party identifiers P_A, P_B.
Initialise: On receiving Init from P_A, and (Init, Δ) from P_B, store the global key $\Delta \in \mathbb{F}_{p^r}$, and ignore all subsequent Init commands.
Extend: On receiving $(\mathsf{Extend}, m, \alpha, \beta)$ from P_A and Extend from P_B, where $m = 2^h$, do:

1. If P_B is honest, sample $\boldsymbol{v} \leftarrow_R \mathbb{F}^m_{p^r}$. Else, receive $\boldsymbol{v}$ from $\mathcal{A}$.
2. Set $\boldsymbol{u} \in \mathbb{F}^m_p$ such that $\{\boldsymbol{u}[i]\}_{i \neq \alpha} = 0$ and $\boldsymbol{u}[\alpha] = \beta$. Compute $\boldsymbol{w} = \boldsymbol{v} + \Delta \cdot \boldsymbol{u} \in \mathbb{F}^m_p$.
3. If P_B is corrupt, receive a set $I \subseteq [0, m)$ from $\mathcal{A}$. If $\alpha \in I$, send $\mathsf{success}$ to P_B and continue. Else, send abort to both parties, output α to P_B and abort.
4. Output $(\boldsymbol{u}, \boldsymbol{w})$ to P_A and $\boldsymbol{v}$ to P_B.

Global-key query: If P_A is corrupted, receive $(\mathsf{guess}, \Delta')$ from $\mathcal{A}$ with $\Delta' \in \mathbb{F}_{p^r}$. If $\Delta' = \Delta$, send $\mathsf{success}$ to P_A and ignore any subsequent global-key query. Else, send abort to both parties and abort.

Figure 4.12: Functionality for a chosen-input sVOLE

as the input. The outputs from $\mathcal{F}_{\mathsf{OT}}$ give P_A $(\boldsymbol{w}[i])_{i \neq \alpha}$ as $\boldsymbol{w}[i] = \boldsymbol{v}[i]$ for $i \neq \alpha$. The only thing that remains is to obtain $\boldsymbol{w}[\alpha] = \boldsymbol{v}[\alpha] + \Delta \cdot \beta$. Recall that parties already have $\langle \alpha \rangle$. Therefore, P_B can send $\mathsf{K}[\beta] - \sum_{i=1}^{m} \boldsymbol{v}[i]$ to P_A, which can compute $\boldsymbol{w}[\alpha]$ as:

$$\boldsymbol{w}[\alpha] = \mathsf{M}[\beta] - (\mathsf{K}[\beta] - \sum_{i=1}^{m} \boldsymbol{v}[i]) - \sum_{i \neq \alpha} \boldsymbol{v}[i] = \mathsf{M}[\beta] - \mathsf{K}[\beta] + \boldsymbol{v}[\alpha] = \boldsymbol{v}[\alpha] + \Delta \cdot \beta$$

To check for malicious behaviour, we run the consistency check from Wolverine, which is described here for completeness. The idea is for parties to samples uniformly random values $\chi_0, \ldots, \chi_{n-1} \in \mathbb{F}_{p^r}$ and checking the randomised version of the VOLE as:

$$\sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{w}[i] = \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{v}[i] + \Delta \cdot \beta \cdot \chi_\alpha$$

P_B cannot compute this however, as it does not know α, β. Therefore, parties can use $\mathcal{F}_{\mathsf{sVOLE}}$ to generate $Z, Y \in \mathbb{F}_{p^r}$ such that $Z = Y + \Delta \cdot \beta \cdot \chi_\alpha$. Since $\beta \cdot \chi_\alpha$ lies in $\mathbb{F}_{p^r}$ as opposed to $\mathbb{F}_p$, we cannot directly use $\mathcal{F}_{\mathsf{sVOLE}}$. Instead, χ_α can be viewed as $(\chi_{\alpha,0}, \ldots, \chi_{\alpha,r-1} \in \mathbb{F}^r_p$. They can then use r calls to $\mathcal{F}_{\mathsf{sVOLE}}$ to which gives P_A $\boldsymbol{z}$ and P_B $\boldsymbol{y}$ such that $\boldsymbol{z} = \boldsymbol{y} + \Delta \cdot \beta \cdot \chi_\alpha$. Let $Z = \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i$ and $Y = \sum_{i=0}^{r-1} \boldsymbol{y}[i] \cdot \mathsf{X}^i$. This means P_A can compute $V_A = \sum_{i=0}^{n-1} \chi_i \boldsymbol{w}[i] - Z$ and P_B can compute $V_B = \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{v}[i] - Y$. The final step is to call $\mathcal{F}_{\mathsf{EQ}}$ with V_A, V_B, which returns either $\mathsf{success}$ or abort.

Theorem 4.3. *If G and G' are pseudorandom generators, then $\Pi^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ UC-realises $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ in the $(\mathcal{F}_{\mathsf{sVOLE}}, \mathcal{F}_{\mathsf{EQ}}, \mathcal{F}_{\mathsf{OT}})$-hybrid model. In particular, no PPT environment $\mathcal{Z}$ can distinguish the real-world execution from the ideal-world one except with probability at most $1/p^r + \mathsf{negl}(k)$.*

Proof. The first part deals with the case of a malicious P_A and the second one with a malicious P_B. In each case we construct a PPT simulator $\mathcal{S}$ which is given access to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ that runs the

Protocol $\Pi^{\mathsf{ci}}_{\mathsf{spsVOLE}}$

Parameters: An extension field $\mathbb{F}_{p^r}$, party identifiers P_A, P_B.
Initialise: Executed only once between a pair of parties. P_A sends Init and P_B sends (Init, Δ) to $\mathcal{F}_{\mathsf{sVOLE}}$.
Extend: Can be executed multiple times. P_A has input (α, β), where $\alpha \in [0, n)$, $\beta \in \mathbb{F}_p^*$.

1. P_A and P_B send Extend to $\mathcal{F}_{\mathsf{sVOLE}}$, which returns $(a, c) \in \mathbb{F}_p \times \mathbb{F}_{p^r}$ to P_A and $b \in \mathbb{F}_{p^r}$ to P_B such that $c = \Delta \cdot a + b$.
2. P_A sets $\delta = c$ and sends $a' = \beta - a \in \mathbb{F}_p$ to P_B which computes $\gamma = b - \Delta \cdot a'$, forming $\langle \beta \rangle$. P_A defines $\boldsymbol{u} \in \mathbb{F}_p^m$ as the single-point vector such that $\boldsymbol{u}[\alpha] = \beta$.
3. P_B samples $s \leftarrow_R \{0,1\}^k$, runs $\mathsf{GGM}(1^m, s)$ to get $(\{v_j\}_{j \in [0,m)}, \{(K_0^i, K_1^i)\}_{i \in [1,h]})$ and sets $\boldsymbol{v}[j] = v_j$ for $j \in [0, m)$. P_A lets $\bar{\alpha_i}$ be the compliment of the ith bit of the binary representation of α. For $i \in [1, h]$, P_A sends $\bar{\alpha_i} \in \{0,1\}$ to $\mathcal{F}_{\mathsf{OT}}$ and P_B sends (K_0^i, K_1^i) to $\mathcal{F}_{\mathsf{OT}}$. P_A receives $K^i_{\bar{\alpha_i}}$, which then runs $\{v_j\}_{j \neq \alpha} = \mathsf{GGM}'(\alpha, \{K^i_{\bar{\alpha_i}}\}_{i \in [1,h]})$.
4. P_B sends $d = \gamma - \sum_{i \in [0,m)} \boldsymbol{v}[i] \in \mathbb{F}_p^r$ to P_A. Then, P_A defines $\boldsymbol{w} \in \mathbb{F}_{p^r}^m$ as the vector with $\boldsymbol{w}[i] = v_i$ for $i \neq \alpha$ and $\boldsymbol{w}[\alpha] = \delta - \left(d + \sum_{i \neq \alpha} \boldsymbol{w}[i]\right)$. Note that $\boldsymbol{w} = \Delta \cdot \boldsymbol{u} + \boldsymbol{v}$.

Consistency check:

1. Both parties send (Extend, r) to $\mathcal{F}_{\mathsf{sVOLE}}$, which returns $(\boldsymbol{x}, \boldsymbol{z}) \in \mathbb{F}_p^r \times \mathbb{F}_{p^r}^r$ to P_A and $\boldsymbol{y}^* \in \mathbb{F}_{p^r}^r$ to P_B such that $\boldsymbol{z} = \Delta \cdot \boldsymbol{x} + \boldsymbol{y}^*$.
2. P_A samples $\chi_i \leftarrow_R \mathbb{F}_{p^r}$ for $i \in [0, m)$ and writes $\chi_\alpha = \sum_{i=0}^{r-1} \chi_{\alpha,i} \cdot \mathsf{X}^i$. Let $\chi_\alpha = (\chi_{\alpha,0}, \ldots, \chi_{\alpha,r-1}) \in \mathbb{F}_p^r$. P_A then computes $\boldsymbol{x}^* = \beta \cdot \boldsymbol{\chi}_\alpha - \boldsymbol{x} \in \mathbb{F}_p^r$ and sends $(\{\chi_i\}_{i \in [0,m)}, \boldsymbol{x}^*)$ to P_B, which computes $\boldsymbol{y} = \boldsymbol{y}^* - \Delta \cdot \boldsymbol{x}^* \in \mathbb{F}_{p^r}^r$.
3. P_A computes $\mathcal{Z} = \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i \in \mathbb{F}_{p^r}$ and $V_A = \sum_{i=0}^{m-1} \chi_i \cdot \boldsymbol{w}[i] - \mathcal{Z} \in \mathbb{F}_{p^r}$, while P_B computes $\mathcal{Y} = \sum_{i=1}^{r-1} \boldsymbol{y}[i] \cdot \mathsf{X}^i \in \mathbb{F}_{p^r}$ and $V_B = \sum_{i=0}^{m-1} \chi_i \cdot \boldsymbol{v}[i] - \mathcal{Y} \in \mathbb{F}_{p^r}$. Then P_A sends V_A to $\mathcal{F}_{\mathsf{EQ}}$, and P_B sends V_B to $\mathcal{F}_{\mathsf{EQ}}$. If either party receives false or abort from $\mathcal{F}_{\mathsf{EQ}}$, it aborts.
4. P_A outputs $(\boldsymbol{u}, \boldsymbol{w})$ and P_B outputs $\boldsymbol{v}$.

Figure 4.13: Protocol for single-point sVOLE

$\mathcal{A}$ as a subroutine and emulates the functionalities $\mathcal{F}_{\mathsf{sVOLE}}, \mathcal{F}_{\mathsf{EQ}}, \mathcal{F}_{\mathsf{OT}}$. We implicitly assume that the simulator $\mathcal{S}$ passes all the communication between the $\mathcal{A}$ and the environment $\mathcal{Z}$.

The $\mathcal{S}$ for a malicious P_A behaves exactly the same as it does in Wolverine [WYKW21b]. The interesting case is when P_B is malicious.

Malicious P_A: Every time the extend procedure is run with inputs (m, α, β), $\mathcal{S}$ interacts with $\mathcal{A}$ as follows:

1. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{sVOLE}}$ and records the values (a, c) that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{sVOLE}}$. When $\mathcal{A}$ sends the message $a' \in \mathbb{F}_p$, then $\mathcal{S}$ sets $\beta = a' + a \in \mathbb{F}_p$ and $\delta = c$.

2. For $i \in [1, h)$, $\mathcal{S}$ samples $K^i \leftarrow_R \{0,1\}^\kappa$; it also samples $K^h \leftarrow_R \mathbb{F}_{p^r}$. Then for $i \in [1, h]$, $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{OT}}$ by receiving $\bar{\alpha}_i \in \{0,1\}$ from $\mathcal{A}$, and returning $K^i_{\bar{\alpha}_i} = K^i$ to $\mathcal{A}$. It sets $\alpha = \alpha_1 \ldots \alpha_h$ and defines $\boldsymbol{u} \in \mathbb{F}_p^m$ as the vector that is 0 everywhere except that $\boldsymbol{u}[\alpha] = \beta$. Next, $\mathcal{S}$ computes $\{v_j\}_{j \neq \alpha} = \mathsf{GGM}'(\alpha, \{K^i_{\bar{\alpha}_i}\}_{i \in [1,h]})$.

3. $\mathcal{S}$ picks $d \leftarrow_R \mathbb{F}_{p^r}$ and sends it to $\mathcal{A}$. Then $\mathcal{S}$ defines $\boldsymbol{w}$ as the vector of length m with $\boldsymbol{w}[i] = v_i$ for $i \neq \alpha$ and $\boldsymbol{w}[\alpha] = \delta - (d + \sum_{i \neq \alpha} \boldsymbol{w}[i])$.

4. $\mathcal{S}$ emulates $\mathcal{F}_{\mathsf{sVOLE}}$ by recording $(\boldsymbol{x}, \boldsymbol{z})$ from $\mathcal{A}$.

5. $\mathcal{S}$ receives $\{\chi_i\}_{i \in [0,n)}$ and $\boldsymbol{x}^* \in \mathbb{F}_p^r$ from $\mathcal{A}$, and sets $\boldsymbol{x}' = \boldsymbol{x}^* + \boldsymbol{x} \in \mathbb{F}_p^r$ and $x' = \sum_{i=0}^{r-1} \boldsymbol{x}'[i] \cdot \mathsf{X}^i$.

6. $\mathcal{S}$ records $V_A \in \mathbb{F}_p^r$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{EQ}}$. It then computes $V'_A = \sum_{i=0}^{m-1} \chi_i \cdot \boldsymbol{w}[i] - \sum_{i=0}^{r-1} \boldsymbol{z}[i] \cdot \mathsf{X}^i \in \mathbb{F}_p^r$ and does:

    a) If $x' = \beta \cdot \chi_\alpha$, then $\mathcal{S}$ checks whether $V_A = V'_A$. If so, $\mathcal{S}$ sends true to $\mathcal{A}$, and sends $\boldsymbol{u}, \boldsymbol{w}$ to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$. Else, $\mathcal{S}$ sends abort to $\mathcal{A}$ and aborts.

    b) Else, $\mathcal{S}$ computes $\Delta' = (V'_A - V_A)/(\beta \cdot \chi_\alpha - x') \in \mathbb{F}_p^r$ and sends a global-key query $(\mathsf{guess}, \Delta')$ to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$. If $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ returns $\mathsf{success}$, $\mathcal{S}$ sends true to $\mathcal{A}$, and sends $\boldsymbol{u}, \boldsymbol{w}$ to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$. Else, $\mathcal{S}$ sends abort to $\mathcal{A}$ and aborts.

7. Whenever $\mathcal{A}$ sends a global-key query to $(\mathsf{guess}, \Delta')$ to the functionality $\mathcal{F}_{\mathsf{sVOLE}}$, $\mathcal{S}$ forwards the query to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ and returns the answer to $\mathcal{A}$. If the answer is abort, $\mathcal{S}$ aborts.

□

Malicious P_B: The simulator $\mathcal{S}$ interacts with $\mathcal{A}$ as follows. First, it simulates the initialisation step by recording the global-key $\Delta \in \mathbb{F}_{p^r}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{sVOLE}}$. Then, every time (Extend, m) is called, $\mathcal{S}$ does:

1. $\mathcal{S}$ records $b \in \mathbb{F}_{p^r}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{sVOLE}}$. Then $\mathcal{S}$ samples $a' \leftarrow_R \mathbb{F}_p$ and sends it to $\mathcal{A}$. Next, $\mathcal{S}$ computes $\gamma = b - \Delta \cdot a'$, and samples $\beta \in \mathbb{F}_p^*$ and sets $\delta = \gamma + \Delta \cdot \beta$.

2. $\mathcal{S}$ records the values $\{(K^i_0, K^i_1)\}_{i \in [1,h]}$ sent to $\mathcal{F}_{\mathsf{OT}}$ by $\mathcal{A}$.

3. $\mathcal{S}$ receives $d \in \mathbb{F}_{p^r}$ from $\mathcal{A}$. Then, for each $\alpha \in [0, n)$, it computes a vector $\boldsymbol{w}_\alpha$ as follows:

    a) Execute $\{v^\alpha_j\}_{j \neq \alpha} = \mathsf{GGM}'(\alpha, \{K^i_{\bar{\alpha}_i}\}_{i \in [1,h]})$ and set $\boldsymbol{w}_\alpha[i] = v^\alpha_i$ for $i \neq \alpha$.

    b) Compute $\boldsymbol{w}_\alpha[\alpha] = \delta - (d + \sum_{i \neq \alpha} \boldsymbol{w}_\alpha[i])$.

4. $\mathcal{S}$ records the vector $\boldsymbol{y}^*$ sent to $\mathcal{F}_{\mathsf{sVOLE}}$ by $\mathcal{A}$.

5. $\mathcal{S}$ samples $\chi_i \leftarrow_R \mathbb{F}_{p^r}$ for $i \in [0, n)$ and $\boldsymbol{x}^* \leftarrow_R \mathbb{F}_{p^r}$, and sends them to $\mathcal{A}$. Then $\mathcal{S}$ computes $\boldsymbol{y} = \boldsymbol{y}^* - \Delta \cdot x^*$.

6. $\mathcal{S}$ computes $Y = \sum_{i=0}^{r-1} \boldsymbol{y}[i] \cdot \mathsf{X}^i$. It then records V_B sent to $\mathcal{F}_{\mathsf{EQ}}$ by $\mathcal{A}$. Then, $\mathcal{S}$ computes a set $I \subseteq [0, n)$ as follows:

    a) For $\alpha \in [0, n)$, compute $V^\alpha_A = \sum_{i=0}^{n-1} \chi_i \cdot \boldsymbol{w}_\alpha[i] - \Delta \cdot \beta \cdot \chi_\alpha - Y$.

    b) Define $I = \{\alpha \in [0, n) | V^\alpha_A = V_B\}$.

$\mathcal{S}$ sends I to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$. If it returns $(\mathsf{abort}, \alpha^*)$, where α^* was the value used by an honest P_A, $\mathcal{S}$ uses α^* to compute the correct $V_A^{\alpha^*}$ and sends $(\mathsf{false}, V_A^{\alpha^*})$ to $\mathcal{A}$ on behalf of $\mathcal{F}_{\mathsf{EQ}}$, and then aborts. Else, $\mathcal{S}$ sends (true, V_B) to $\mathcal{A}$.

7. $\mathcal{S}$ chooses an arbitrary $\alpha \in I$ and computes a vector $\boldsymbol{v}$ as follows:

   a) Set $\boldsymbol{v}[i] = \boldsymbol{w}_\alpha[i]$ for $i \in [0, n)_{i \neq \alpha}$.

   b) Set $\boldsymbol{v}[\alpha] = \gamma - d - \sum_{i \neq \alpha} \boldsymbol{v}[i]$.

   $\mathcal{S}$ sends $\boldsymbol{v}$ to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ and outputs whatever $\mathcal{A}$ outputs.

We first consider the view of the adversary $\mathcal{A}$ in the ideal-world execution and the real-world execution. The values a' and $\boldsymbol{x}^*$ simulated by $\mathcal{S}$ have the same distribution as the real values, which are masked by a uniformly random element/vector output by $\mathcal{F}_{\mathsf{sVOLE}}$. The set I extracted by $\mathcal{S}$ corresponds to a selective failure attack on the output index α^* of P_A. If $\mathcal{S}$ receives an abort from $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$, it means $\alpha^* \notin I$. In the real protocol, P_A aborts if $V_A^{\alpha^*} \neq V_B$. Therefore, $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ only aborts if the real-world protocol aborts.

Since α is given as input by P_A instead of being chosen at random, $\mathcal{S}$ cannot pick a random $\alpha \in [0, n) \setminus I$, as it does in [WYKW21b]. It needs to send the V_A that corresponds to the V_A that an honest P_A would have sent in the real-world. In order to facilitate this, the $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ functionality is designed to return the α^* that was used in the real protocol, in the case of an abort. This means the distribution of V_A sent by the $\mathcal{S}$ is indistinguishable from the real world distribution.

From $\Pi^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ to $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$

The final step is to go from single-point VOLE to standard (programmable) VOLE. Here, we will realize $\mathcal{F}^{\mathsf{prog}}_{\mathsf{VOLE}}$ instantiated with a particular expansion function $\mathsf{Expand} : S \to \mathbb{F}_p^m$, based on a variant of the LPN assumption.

t-regular vector: A t-regular vector $\boldsymbol{e}$ is defined as a set of t vectors $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_t$ concatenated, wherein each $\boldsymbol{e}_i$ is a sparse vector with Hamming weight one.

We use the dual form of LPN over $\mathbb{F}_p$, with a regular error distribution. This has also been considered in previous works [BCG+19a, WYKW21b].

Definition 4.1 (Regular Dual-LPN assumption). *Let $H \in \mathbb{F}_p^{k \times m}$, and consider the following game $G_b(\kappa)$ with a PPT adversary $\mathcal{A}$, parameterised by a bit b and the security parameter κ:*

1. *Sample a random, t-regular vector $\boldsymbol{e} \in \mathbb{F}_p^k$.*

2. *If $b = 1$, let $\boldsymbol{y} = H \cdot \boldsymbol{e}$, else sample $\boldsymbol{y} \leftarrow_R \mathbb{F}_p^m$.*

3. *Send $\boldsymbol{y}$ to $\mathcal{A}$, which then outputs a bit b' (in case of abort, define the output of $\mathcal{A}$ to be $\perp$).*

The assumption states that $\mathcal{A}$ has negligible advantage in distinguishing $G_0(\kappa)$ and $G_1(\kappa)$.

Functionality $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$

Parameters: Finite field $\mathbb{F}_{p^r}$, and expansion function $\mathsf{Expand} : S \to \mathbb{F}_p^m$ with seed space S and output length m.
The functionality runs between parties P_A and P_B.
Initialise: On receiving Init from P_A, and (Init, Δ) from P_B, store Δ, and ignore all subsequent Init commands.
Extend: On receiving Extend from P_B and $(\mathsf{Extend}, \mathsf{seed})$ from P_A, where $\mathsf{seed} \in S$:

1. Compute $\boldsymbol{u} = \mathsf{Expand}(\mathsf{seed})$.
2. Sample $\boldsymbol{v} \leftarrow_R \mathbb{F}_{p^r}^m$ and compute $\boldsymbol{w} = \boldsymbol{u} \cdot \Delta + \boldsymbol{v}$.
3. If P_B is corrupt, receive a set I from $\mathcal{A}$. If $\mathsf{seed} \in I$, send $\mathsf{success}$ to P_B and continue. Else, send abort to both parties, output seed to P_B and abort.
4. Output $(\boldsymbol{u}, \boldsymbol{w})$ to P_A and $\boldsymbol{v}$ to P_B.

Corrupt parties: If P_B is corrupt, $\boldsymbol{v}$ may be chosen by $\mathcal{A}$. For a corrupt P_A, $\mathcal{A}$ can choose $\boldsymbol{w}$ (and then $\boldsymbol{v}$ is recomputed accordingly).
Global key query: If P_A is corrupted, receive $(\mathsf{guess}, \Delta')$ from $\mathcal{A}$ with $\Delta' \in \mathbb{F}_{p^r}$. If $\Delta' = \Delta$, send $\mathsf{success}$ to P_A and ignore any subsequent global key query. Else, send abort to both parties and abort.

Figure 4.14: Functionality for programmable VOLE

Expansion function: Fix a dual-LPN matrix $H \in \mathbb{F}_p^{m \times k}$. We consider a seed space $S \subset \mathbb{F}_p^k$ consisting of t-regular vectors in $\mathbb{F}_p^k$. We define the LPN-based expand function

$$\mathsf{Expand}^{\mathsf{LPN}} : S \to \mathbb{F}_p^m, \qquad \mathsf{Expand}^{\mathsf{LPN}}(\boldsymbol{e}) = H \cdot \boldsymbol{e}$$

Overview of $\Pi_{\mathsf{VOLE}}^{\mathsf{prog}}$: The first step is to execute $\mathcal{F}_{\mathsf{sVOLE}}$, which gives Δ to P_B on Init and gives $\langle \boldsymbol{u} \rangle \in \mathbb{F}_p^k$. In addition, they run $\mathcal{F}_{\mathsf{spsVOLE}}^{\mathsf{ci}}$ t times, to get vectors of authenticated values. Vectors are denoted by $\boldsymbol{e}_i$, each of them is of length m/t and has exactly one nonzero entry. Parties use the public matrix H to convert these to a vector of authenticated values of length m.

Under the regular dual-LPN assumption, the values appear pseudorandom to P_B, if the seed S was sampled at random. Note, however, that the protocol perfectly realizes $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ without relying on dual-LPN, because $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ itself is defined in terms of the expansion function. Therefore, it is only when using $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}$ to instantiate our preprocessing protocol, where LPN comes into play.

4.4 Dynamic SPDZ

We now show how to use our preprocessing to obtain a dynamic variant of the SPDZ protocol [DPSZ12, DKL+13]. The preprocessing is performed between the entire set of parties $\mathcal{P}_{\mathsf{main}} = \{P_1, \ldots, P_n\}$, and later, when an *online phase committee* $\mathcal{P}_{\mathsf{curr}} \subset \mathcal{P}_{\mathsf{main}}$ wants to run MPC, they non-interactively select the relevant preprocessing data, and run our online phase. We consider evaluating arithmetic circuits over $\mathbb{F}_p$ for a large enough (superpolynomial) p, and will use $\mathcal{F}_{\mathsf{Prep}}$ entirely over $\mathbb{F}_p$ (i.e. not using the extension field $\mathbb{F}_{p^r}$).

Protocol $\Pi^{\mathsf{prog}}_{\mathsf{VOLE}}$

Parameters: Extension field $\mathbb{F}_{p^r}$, length m, noise weight t, LPN dimension n and matrix $H \in \mathbb{F}^{m \times k}$, and party identifiers P_A, P_B. $q = k/t$.

Intialise: Executed only once between two parties. P_A, P_B send Init, (Init, Δ) respectively to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$.

Extend: On input seed from P_A, where seed describes a t-regular vector $\boldsymbol{e} \in \mathbb{F}_p^k$:

1. For $i \in [1, t]$, P_A and P_B send (Extend, q) to $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$, which returns $(\boldsymbol{e}_i, \boldsymbol{c}_i)$ to P_A and $\boldsymbol{b}_i$ to P_B such that $\boldsymbol{c}_i = \Delta \cdot \boldsymbol{e}_i + \boldsymbol{b}_i \in \mathbb{F}_{p^r}^q$ and $\boldsymbol{e}_i \in \mathbb{F}_p^q$ has exactly one nonzero entry. If either party receives abort from $\mathcal{F}^{\mathsf{ci}}_{\mathsf{spsVOLE}}$ in any of these executions, it aborts.
2. P_A defines $\boldsymbol{e} = (\boldsymbol{e}_1, \ldots, \boldsymbol{e}_t) \in \mathbb{F}_p^k$ and $\boldsymbol{c} = (\boldsymbol{c}_1, \ldots, \boldsymbol{c}_t) \in \mathbb{F}_{p^r}^k$. Then P_A computes $\boldsymbol{x} = H \cdot \boldsymbol{e}$ and $\boldsymbol{z} = H \cdot \boldsymbol{c}$. P_B defines $\boldsymbol{b} = (\boldsymbol{b}_1, \ldots, \boldsymbol{b}_t) \in \mathbb{F}_{p^r}^k$ and computes $\boldsymbol{y} = H \cdot \boldsymbol{b} \in \mathbb{F}_{p^r}^m$.
3. P_A outputs $(\boldsymbol{s}, \boldsymbol{M}[\boldsymbol{s}]) = (\boldsymbol{x}, \boldsymbol{z}) \in \mathbb{F}_p^m \times \mathbb{F}_{p^r}^m$. P_B updates $\boldsymbol{v}$ by setting $\boldsymbol{v} = \boldsymbol{y} \in \mathbb{F}_{p^r}^m$, and outputs $\boldsymbol{K}[\boldsymbol{s}] = \boldsymbol{y} \in \mathbb{F}_{p^r}^m$.

Figure 4.15: Protocol to extend $\mathsf{spsVOLE}$

Since our preprocessing is significantly weaker than SPDZ — due to faulty and partially authenticated triples — we cannot use the same online phase for multiplications. Instead, in our multiplication protocol, we will first have the parties add a MAC to the 'c' component of a triple (using a preprocessed random authenticated value), and then use the fully authenticated triple to multiply. Since the triples may be faulty, to verify multiplications we take the approach of [CGH$^+$18], where parties compute two versions of the circuit: one with the actual inputs and one with a randomised version of the inputs. At the end of the protocol, they first run a MAC Check protocol to verify correctness of the opened values in multiplication, as in SPDZ. If this check succeeds, they open the random value used to compute the randomised circuit. Using that, they take a random linear combination of wires in both circuits and check that they are the consistent. We start by describing the online phase protocol $\Pi_{\mathsf{SPDZ\text{-}Online}}$, before analysing the verification process and concluding with a cost analysis.

SPDZ Sharing, Share Conversion and Opening.

A SPDZ share of $v \in \mathbb{F}_p$ contains a vector of additive shares $([v], [\Delta], [\Delta \cdot v])$, where the shares are held by each P_i within the current committee $\mathcal{P}_{\mathsf{curr}}$. We denote this by $[\![\cdot]\!]^{\mathcal{P}_{\mathsf{curr}}}$, and omit $\mathcal{P}_{\mathsf{curr}}$ when it is clear from context. Note that the MAC key Δ is fixed for every sharing in the same committee.

Given a pairwise authenticated sharing $\langle x \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}}}$, the parties can *locally* convert this into a SPDZ sharing with the procedure Π_{Convert}:

$$\Pi_{\mathsf{Convert}}(\langle x \rangle^{\mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}}}) : P_i \text{ outputs } \Big(x^i, \Delta^i, \Delta^i \cdot x^i + \sum_{j \in \mathcal{P}_{\mathsf{curr}}} (M_j^i - K_j^i) \Big)$$

Protocol $\Pi_{\mathsf{SPDZ\text{-}MAC}}$

Usage: Parties in $\mathcal{P}_{\mathsf{curr}}$ want to check the MACs on opened values $(A_1, \ldots, A_m)$.

1. Parties in $\mathcal{P}_{\mathsf{curr}}$ call $\mathcal{F}_{\mathsf{Rand}}$ to obtain random values $\chi_1, \ldots, \chi_m \in \mathbb{F}_p$.
2. Compute $A = \sum_{j=1}^{m} \chi_j \cdot A_j$ and $[\gamma] = \sum_{j=1}^{m} \chi_j \cdot [\Delta \cdot A_j]$.
3. Compute $[\sigma] = [\gamma] - [\Delta] \cdot A$. Each $P_i \in \mathcal{P}_{\mathsf{curr}}$ calls $\mathcal{F}_{\mathsf{Commit}}$ with input $[\sigma]$.
4. Parties open their commitments and check that $\sum_{i=1}^{n} [\sigma] = 0$. If not, output abort, else output $\mathsf{continue}$.

Figure 4.16: Protocol to check MACs in Dynamic SPDZ

where M_j^i, K_j^i are P_i's MACs and MAC keys from the $\langle\cdot\rangle$ sharing. By inspection, this gives a consistent sharing $[\![x]\!]^{\mathcal{P}_{\mathsf{curr}}}$.

We let Π_{Open} denote the opening protocol, which given $[\![x]\!]$ or $[x]$ has all parties send to each other their shares x^i and reconstruct $x = \sum x^i$. This procedure does not check the MACs, so it may be unreliable. To check the MAC on an opened value (after running Π_{Open}), we use the standard SPDZ MAC check protocol [DKL+13], shown in Fig. 4.16.

Online Protocol.

$\Pi_{\mathsf{SPDZ\text{-}Online}}$ (Fig. 4.17) begins with each P_i in a set of parties $\mathcal{P}_{\mathsf{curr}} \subseteq \mathcal{P}_{\mathsf{main}}$ querying $\mathcal{F}_{\mathsf{Prep}}$ to receive an authenticated random value $\langle t\rangle$, where P_i knows t, and every other party has a share of the MAC. P_i uses this to generate $[\![\cdot]\!]$ sharing of its input x. This takes one round, where P_i sends $x + t$ to everyone else, along with a fresh sharing of x. The parties then use their MACs from $\langle t\rangle$ to obtain the MAC share for $[\![x]\!]$. For the randomised circuit evaluation (used to check multiplications), during initialization the parties first use $\mathcal{F}_{\mathsf{Prep}}$ to obtain a random sharing $[\![r]\!]$. Then, whenever an input $[\![x]\!]$ is authenticated, the parties multiply it with $[\![r]\!]$, using a triple from $\mathcal{F}_{\mathsf{Prep}}$.

Addition and multiplication by a public constant are standard operations, performed locally by every party on its shares. Multiplication is the more challenging operation as we do not have fully authenticated triples. The first step is to call $\mathcal{F}_{\mathsf{Prep}}$ twice to get two triples $([\![a]\!], [\![b]\!], [c])$, $([\![a']\!], [\![b']\!], [c'])$, as well as two random values $[\![l]\!], [\![l']\!]$, incrementing the corresponding counter after each call. $[\![l]\!], [\![l']\!]$ are used to authenticate $[c], [c']$ of the triples. This is done by computing $[l + c], [l' + c']$ locally, and opening the values by broadcasting the shares. Parties can then locally compute the MAC on c as $\Delta^i \cdot (l + c) - [\Delta \cdot l]$ for P_i. However, since we do not check the correctness at this point, the MACs in $[\![c]\!], [\![c']\!]$ might have an additive error chosen by the adversary. In addition, the c part of the triple may have errors, since this is allowed by $\mathcal{F}_{\mathsf{Prep}}$.

Let P_i be an honest party in $\mathcal{P}_{\mathsf{curr}}$. In a triple (a, b, c), c^i can have additive errors of the form $\{\delta_a^{j,i} \cdot b^i + \delta_b^{j,i} \cdot a^i\}_{j \in \mathcal{P}_{\mathcal{A}}}$, where $\delta_a^{j,i}, \delta_b^{j,i}$ are chosen by a malicious P_j in $\mathcal{F}_{\mathsf{Prep}}$. We show in Lemma 4.3 that these errors do not give the adversary any additional power compared to injecting additive errors to the output of multiplications in the online phase, and will be detected by our verification procedure. Using the potentially inconsistent triples, parties then compute the multiplications $x \cdot y$, $rx \cdot y$ by opening $[\![x - a]\!], [\![y - b]\!], [\![rx - a']\!], [\![y - b']\!]$ in the standard way of using Beaver triples. To open $[\![\cdot]\!]$-shared values, parties broadcast arithmetic shares of the value and continue with the computation. At the end of the protocol, the verification phase computes a MAC Check on all the authenticated values that had been opened. The protocol for the online phase of Dynamic SPDZ appears in Fig. 4.17.

Note that for a multiplication $x \cdot y$, it is important that $[l + c]$ is not opened in the same round as $[\![x - a]\!], [\![y - b]\!]$. This is because if we do, a rushing adversary can perform the following attack: To make the illustration simpler, we consider only two parties P_i, P_j in the committee. Suppose the adversary P_j introduces an error $\delta_b^{j,i} \cdot a^i$ with an honest party P_i, using the errors in $\mathcal{F}_{\mathsf{Prep}}$. The adversary then waits until it receives $x - a$, and when opening $[l + c]$, injects another additive error given by $\big((x - a) + a^j\big) \cdot \delta_b^{j,i}$. Therefore, the triple will now be:

$$\begin{aligned}[\![a]\!], [\![b]\!], [\![c]\!] &= \{[c] + \delta_b^{j,i} \cdot a^i + [(x - a) + a^j] \cdot \delta_b^{j,i}, [\Delta \cdot c]\} \\ &= \{[c] + x \cdot \delta_b^{j,i}, [\Delta \cdot c]\}\end{aligned}$$

This results in the adversary mounting a selective failure attack, since the error now depends on the secret wire value x. It can be avoided by making the adversary add the additive error prior to learning $x - a$. A simple way of achieving this is to authenticate c one round prior to opening $x - a$. Although this costs an additional round, the authentication step of a triple for the current layer can easily be merged with the opening of $x - a$ from the previous layer. This is still secure because the triples are independent and the adversary does not gain anything by opening the independently masked c in the previous layer.

The verification phase, described in Fig. 4.18, is run before outputting any result of a computation. First, the parties check the MACs on all the values that were opened over the course of the computation. If the check fails, the parties abort. Otherwise, they proceed by checking correctness of multiplications, with the check from [CGH$^+$18], which involves checking a random linear combination of the inputs and outputs, and randomised versions of them. Parties start by calling $\mathcal{F}_{\mathsf{Coin}}$ to receive random challenges $\alpha_1, \ldots, \alpha_N$ and $\beta_1, \ldots, \beta_M \in \mathbb{F}_p$. They locally compute $[\![u]\!] = \sum_{i=1}^{N} \alpha_i \cdot [\![rz_i]\!] + \sum_{i=1}^{M} \beta_i \cdot [\![\alpha v_i]\!]$ and $[\![w]\!] = \sum_{i=1}^{N} \alpha_i \cdot [\![z_i]\!] + \sum_{i=1}^{M} \beta_i \cdot [\![v_i]\!]$. If no cheating had occurred, opening $[\![u]\!] - r \cdot [\![w]\!]$ should result in zero. To check this, parties securely reconstruct $[\![r]\!]$ using Π_{Open}, locally compute $[\![u]\!] - r \cdot [\![w]\!]$. If the opened value is not zero, they reject.

The analysis of the verification phase proceeds similarly to that of [CGH$^+$18], except we also need to deal with the additional errors from our preprocessing functionality.

Lemma 4.3. *Suppose $\mathcal{A}$ introduces additive errors of the form $\delta_a^{j,i}, \delta_b^{j,i} \neq 0$, for malicious parties P_j and honest P_i in $\mathcal{F}_{\mathsf{Prep}}$, and in $\Pi_{\mathsf{SPDZ\text{-}Online}}$ additive errors $\delta_c, \delta_{c'} \neq 0$ when authenticating triples a, b, c and a', b', c' respectively. If any errors are non-zero, then the Verification phase in $\Pi_{\mathsf{SPDZ\text{-}Online}}$ fails to abort with probability less than $2/p$.*

Proof. Consider a multiplication gate at layer k, wherein the multiplications carried out are $z_k = x_k \cdot y_k$, and $rz_k = rx_k \cdot y_k$. Note that rx, ry will have errors from the layer $k - 1$. $\mathcal{A}$ can insert an additive error when c, c' are authenticated and these are denoted by $\delta_c, \delta_{c'}$ respectively. The errors $\delta_c, \delta_{c'}$ are going to be consistent with the MACs as well, due to the way c and c' are authenticated. They will not get caught during the MAC Check, which is why we compute the randomised circuit in addition to using MACs.

$\mathcal{A}$ can insert an additive error in the output of a multiplication, and the error is indexed by ε^k for layer k. Let the error introduced by $\mathcal{A}$ in computing $[\![r]\!] \cdot [\![x]\!]$ be denoted by ε_1, ignoring the superscript for simplicity. The errors in computing $[\![x]\!] \cdot [\![y]\!]$ and $[\![r]\!] \cdot [\![v]\!]$, where $[\![v]\!]$ is the input, are indicated by ε_2, and ε_4 respectively. Finally, computing $[\![rz]\!]$ is done by computing $[\![rx]\!] \cdot [\![y]\!]$, and the error introduced is ε_3.

In addition, we need to account for the errors in the triples used to carry out these multiplications. Parties receive a triple of the form $[\![a]\!], [\![b]\!], [c]$ from $\mathcal{F}_{\mathsf{Prep}}$ in the online phase. The $[c]$ part of the triple has additive errors due to using an inconsistent $\mathcal{F}_{\mathsf{OLE}}^{\mathsf{prog}}$, as explained in Section 4.3. These errors can be viewed as $[\hat{c}] = [c] + \{\delta_a^{j,i} \cdot b^i + \delta_b^{j,i} \cdot a^i\}$, for $j \in \mathcal{P}_A, i \in \mathcal{P}_H$. On top of this, parties authenticate $[c]$ in the online phase before processing the multiplication gates, wherein $\mathcal{A}$ can introduce another additive error, denoted by δ_c. We let $\hat{\varepsilon} = \varepsilon + \{\delta_a^{j,i} \cdot b^i + \delta_b^{j,i} \cdot a^i\} + \delta_c$, for $j \in \mathcal{P}_{\mathcal{A}}, i \in \mathcal{P}_H$ and a multiplication that used a triple $[\![a]\!], [\![b]\!], [\![c]\!]$. Therefore, the values computed will be:

1. $[\![rx]\!] = [\![r]\!] \cdot [\![x]\!] \rightarrow [\![rx + \hat{\varepsilon}_1]\!]$ (layer $k-1$)
2. $[\![z]\!] = [\![x]\!] \cdot [\![y]\!] \rightarrow [\![xy + \hat{\varepsilon}_2]\!]$
3. $[\![rz]\!] = [\![rx]\!] \cdot [\![y]\!] \rightarrow ([\![rx + \hat{\varepsilon}_1]\!] \cdot [\![y]\!]) + \hat{\varepsilon}_3$
4. $[\![rv]\!] = [\![r]\!] \cdot [\![v]\!] \rightarrow [\![rv + \hat{\varepsilon}_4]\!]$

Note: The MACs on these values have been checked for consistency by this point (and we ignore here the case that an invalid MAC was successfully forged).

Parties sample random values $\alpha_1, \ldots, \alpha_N$ and $\beta_1, \ldots, \beta_M$ to compute a random linear combination on the actual values and their randomised variants. This is computed for all the inputs to the circuit, and the outputs of every multiplication gate. The random linear combination of the actual values is denoted by $[\![w]\!]$ and the randomised one is denoted by $[\![u]\!]$. The idea is that parties will then open $[\![r]\!]$, and compute $[\![u]\!] - r \cdot [\![w]\!]$. Ideally, this value would be equal to 0. We calculate and show that the probability that $\mathcal{A}$ injects errors as detailed earlier, and does not get caught in the check is upper bounded by $2/p$.

Parties start by computing $[\![u]\!], [\![w]\!]$ as,

$$[\![u]\!] = \sum_{i=1}^{N} \alpha_i \cdot \big((rx + \hat{\varepsilon}_1^i) \cdot y + \hat{\varepsilon}_3^i\big) + \sum_{i=1}^{M} \beta_i \cdot (rv + \hat{\varepsilon}_4^i)$$

$$[\![w]\!] = \sum_{i=1}^{N} \alpha_i \cdot (x \cdot y + \hat{\varepsilon}_2^i) + \sum_{i=1}^{M} \beta_i \cdot v$$

$$\begin{aligned}
[\![u]\!] - r \cdot [\![w]\!] &= \sum_{i=1}^{N} \alpha_i \cdot \big((rx + \hat{\varepsilon}_1^i) \cdot y + \hat{\varepsilon}_3^i\big) + \sum_{i=1}^{M} \beta_i \cdot (rv + \hat{\varepsilon}_4^i) \\
&- r \cdot \left(\sum_{i=1}^{N} \alpha_i \cdot (x \cdot y + \hat{\varepsilon}_2^i) + \sum_{i=1}^{M} \beta_i \cdot v\right) \\
&= \sum_{i=1}^{N} \alpha_i (\hat{\varepsilon}_1^i \cdot y + \hat{\varepsilon}_3^i - r \cdot \hat{\varepsilon}_2^i) + \sum_{i=1}^{M} \beta_i \cdot \hat{\varepsilon}_4^i
\end{aligned}$$

The analysis, below, is similar to [CGH⁺18]. The intuition about why the additional errors introduced in the triples do not give the adversary any additional advantage is as follows. Errors in $[c]$ received from $\mathcal{F}_{\mathsf{Prep}}$ are of the form $\delta_a^{j,i} \cdot b^i$, for when a corrupt P_j interacts with an honest

P_i. Since the adversary does not know the honest P_i's share b^i, this is going to be a random additive error that is not known to $\mathcal{A}$. At this point, if the triple was authenticated in the same round as the computing the multiplication, in other words opening $x - a, y - b$ along with opening $l + c$, $\mathcal{A}$ can wait until it receives $x - a, y - b$ in the clear. Using these values, it can choose a δ_c such that this results in an error of the form $x \cdot \delta_b^{j,i}$, a selective failure attack.

When we later authenticate the triple, $\mathcal{A}$ has still learnt no information about a or b (since we haven't yet opened $x - a, y - b$), so any error δ_c that $\mathcal{A}$ injects will also be an independent, additive error.

The analysis can be split into two cases:

Case 1: There exists some index i such that $\hat{\varepsilon}_4^i \neq 0$. Let m be the smallest one for which it holds. $[\![u]\!] - r \cdot [\![w]\!] = 0$ if and only if:

$$\beta_m = \left(-\sum_{i=1}^{N} \alpha_i \left(\hat{\varepsilon}_1^i \cdot y + \hat{\varepsilon}_3^i - r \cdot \hat{\varepsilon}_2^i \right) + \sum_{i \neq m}^{M} \beta_i \cdot \hat{\varepsilon}_4^i \right) \cdot (\hat{\varepsilon}_4^m)^{-1} \tag{4.1}$$

Since β_m is chosen independently and is uniformly distributed over $\mathbb{F}$, this holds with probability at most $1/p$.

Case 2: All $\hat{\varepsilon}_4^i = 0$, meaning there was no cheating in the triple used to compute $[\![rv]\!]$ or in the output of the multiplication. Assuming the multiplication wires in the succeeding layers were tampered, $\hat{\varepsilon}_2^i \neq 0$ and/or $\hat{\varepsilon}_3^i \neq 0$. Let k be the wire for this, and it holds that $\hat{\varepsilon}_1^k = 0$ for the wire as no input was tampered with before this point. Therefore, $[\![u]\!] - r \cdot [\![w]\!] = 0$ if and only if,

$$\alpha_k \cdot (\hat{\varepsilon}_3^k - r \cdot \hat{\varepsilon}_2^k) = -\sum_{i=1}^{N} \alpha_i \left(\hat{\varepsilon}_1^i \cdot y + \hat{\varepsilon}_3^i - r \cdot \hat{\varepsilon}_2^i \right) \tag{4.2}$$

There are two scenarios, one in which $(\hat{\varepsilon}_3^i - r \cdot \hat{\varepsilon}_2^i) = 0$. The probability of this happening is $1/p$ as r is sampled independently and $\mathcal{A}$ does not know r at the time of injecting errors into the triple or even to the output of the multiplication gate. The other scenario is when $(\hat{\varepsilon}_3^i - r \cdot \hat{\varepsilon}_2^i) \neq 0$. Since α_k is chosen independently and not known to $\mathcal{A}$, the probability of this holding is $(1 - 1/p) \cdot 1/p$. Therefore the total probability of the adversary passing the check in Case 2 is bounded by $2/p$.

$\square$

The following theorem shows that the protocol securely realizes the standard arithmetic black-box functionality, $\mathcal{F}_{\mathsf{ABB}}$ (recall, this is identical to $\mathcal{F}_{\mathsf{DABB}}$ in Fig. 4.5, except the operations are all carried out in one committee, $\mathcal{P}_{\mathsf{curr}}$).

Theorem 4.4. *Protocol* $\Pi_{\mathsf{SPDZ\text{-}Online}}$ *UC-securely computes* $\mathcal{F}_{\mathsf{ABB}}$ *in the presence of a static malicious adversary corrupting up to all-but-one of the parties in* $\mathcal{P}_{\mathsf{curr}}$*, in the* $(\mathcal{F}_{\mathsf{Prep}}, \mathcal{F}_{\mathsf{Coin}})$*-hybrid model.*

Proof. We construct a PPT Simulator ($\mathcal{S}$) that run the adversary ($\mathcal{A}$) as a subroutine, and is given access to $\mathcal{F}_{\mathsf{DABB}}$. It internally emulates the functionalities $\mathcal{F}_{\mathsf{VOLE}}^{\mathsf{prog}}, \mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{Commit}}$ and we implicitly assume that it passes all communication between $\mathcal{A}$ and the environment ($\mathcal{Z}$).

The parties controlled by the $\mathcal{A}$ are indicated by $\mathcal{P}_{\mathcal{A}}$ and the honest parties by $\mathcal{P}_{\mathcal{H}}$. The simulator uses a flag which is set to 1 in case $\mathcal{A}$ is caught cheating before the consistency check happens, and the simulation is carried on. The simulation proceeds as follows:

Malicious $\mathcal{P}_A$:

Init: Receive $(\mathsf{Init}, m_T, m_R)$ from $P_i \in \mathcal{P}_\mathcal{A}$ sent to $\mathcal{F}_{\mathsf{Prep}}$, sample a random Δ^i and send it back. Receive $(\mathsf{Rand}, P_i, \mathcal{P}_{\mathsf{curr}}, \mathsf{rcount})$ from each $P_i \in \mathcal{P}_\mathcal{A}$, abort if $\mathcal{P}_{\mathsf{curr}}$ is not consistent across calls. Receive $\mathcal{A}$'s shares for $[\![r]\!]$ and store them. Sample random shares for inputs of $\mathcal{P}_H$ and send them to $\mathcal{A}$.

Input:

1. Receive $(\mathsf{Rand}, P_i, \mathcal{P}_{\mathsf{curr}}, \mathsf{rcount})$ from each $P_i \in \mathcal{P}_\mathcal{A}$, abort if $\mathcal{P}_{\mathsf{curr}}$ is not consistent across calls. Receive $\mathcal{A}$'s shares for $\langle t \rangle$ and store them. For the honest parties' calls to $\mathcal{F}_{\mathsf{Prep}}$, let $\mathcal{A}$ choose its shares and sample the honest parties' shares at random.

2. Simulate the multiplication step as described below.

Addition, Multiplication by constant: Need not be simulated as they are local operations.

Multiplication:

3. Receive the Trip calls to $\mathcal{F}_{\mathsf{Prep}}$, sample random values for $\mathcal{A}$'s shares of the triples and send them. Receive $\{\delta_a^j\}_{j\in\mathcal{P}_\mathcal{A}}, \{\delta_b^j\}_{j\in\mathcal{P}_\mathcal{A}}$ from $\mathcal{A}$ and if either $\sum_{j\in\mathcal{P}_\mathcal{A}} \delta_a^j \neq 0$ or $\sum_{j\in\mathcal{P}_\mathcal{A}} \delta_b^j \neq 0$, set $\mathsf{flag} = 1$.

4. On receiving the Rand call to $\mathcal{F}_{\mathsf{Prep}}$, sample random values for the shares $[\![l]\!], [\![l']\!]$ and send them to $\mathcal{A}$.

5. Receive shares of $[\![x-a]\!], [\![y-b]\!], [\![rx-a']\!], [\![y-b']\!]$ and $[l+c], [l'+c']$. $\mathcal{S}$ computes the correct shares $\mathcal{A}$ was supposed to send, and if they are inconsistent, sets $\mathsf{flag} = 1$. Send random values for shares of $\mathcal{P}_H$.

6. At this point, one of the following things can happen:

   a) Case 1: The $\mathsf{flag} = 1$ because $\mathcal{A}$ cheated in one of the openings by sending inconsistent values. In this case, $\mathcal{S}$ sends random values on behalf of the honest parties in $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ and aborts at the end of it.

   b) Case 2: The $\mathsf{flag} = 1$ because $\mathcal{A}$ cheated in one of the calls to Trip during a multiplication but not in the openings. $\mathcal{S}$ proceeds with $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ by simulating the Rand call to $\mathcal{F}_{\mathsf{Prep}}$. It then records $\{\sigma^i\}_{\mathcal{P}_\mathcal{A}}$ sent to $\mathcal{F}_{\mathsf{Commit}}$ during $\Pi_{\mathsf{SPDZ\text{-}MAC}}$. If $\mathcal{A}$ sent the correct value, it samples shares for the honest parties such that $\sum_{i=1}^n \sigma^i = 0$ and sends them to $\mathcal{A}$. It then simulates $\Pi_{\mathsf{SPDZ\text{-}Verify}}$ by sending random values for the honest party shares and aborts at the end of it.

   c) Case 3: The $\mathsf{flag} = 0$, but $\mathcal{A}$ cheats in $\Pi_{\mathsf{SPDZ\text{-}MAC}}$. $\mathcal{S}$ aborts at the end of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$.

   d) Case 4: The $\mathsf{flag} = 0$ and there was no cheating in the MACs, so $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ does not abort, but the $\mathcal{A}$ causes an inconsistency in the randomised circuit computation. This could be in one of four places:

      i. Opening of $[\![r]\!]$.
      ii. $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ on r.
      iii. Opening of $[\![u]\!] - r \cdot [\![w]\!]$.
      iv. $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ on $u - rw$.

   In this case, $\mathcal{S}$ records $\{\sigma^i\}_{\mathcal{P}_\mathcal{A}}$ sent to $\mathcal{F}_{\mathsf{Commit}}$ during $\Pi_{\mathsf{SPDZ\text{-}MAC}}$, and samples shares for the honest parties such that $\sum_{i=1}^n \sigma^i = 0$ and sends them to $\mathcal{A}$. In $\Pi_{\mathsf{SPDZ\text{-}Verify}}$, send random values for $[\![r]\!]$, and $[\![u]\!] - r \cdot [\![w]\!]$. Abort at the end of the protocol.

e) Case 5: There was no cheating. $\mathcal{S}$ simulates $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ as in the previous cases when there was no cheating. In $\Pi_{\mathsf{SPDZ\text{-}Verify}}$, it opens a random $[\![r]\!]$ to $\mathcal{A}$ by sending random shares on behalf of $\mathcal{P}_H$. It receives shares of $[\![u]\!] - r \cdot [\![w]\!]$ from $\mathcal{A}$, and samples shares such that $u - r \cdot w = 0$. To compute the outputs, $\mathcal{S}$ sends $\mathcal{A}$'s inputs to $\mathcal{F}_{\mathsf{ABB}}$ using the relevant commands and forwards the output it receives from $\mathcal{F}_{\mathsf{ABB}}$ to $\mathcal{A}$. If $\mathcal{A}$ outputs abort, forward it to $\mathcal{F}_{\mathsf{ABB}}$ and abort.

We now briefly argue that $\mathcal{A}$ cannot distinguish between interacting with the $\mathcal{S}$ and $\mathcal{F}_{\mathsf{Prep}}$, and $\mathcal{F}_{\mathsf{ABB}}$. In the input phase the adversary in both the simulation and the real world, only sees uniformly random values sent by the honest parties since they are masked by a random value not known to $\mathcal{A}$. Addition, addition by a constant, and multiplication by a constant are local operations. In all the calls to $\mathcal{F}_{\mathsf{Prep}}$ using Trip, Rand, $\mathcal{A}$ is allowed to choose its own share therefore the distribution of the MAC shares on these values between the real world and the simulation is perfectly indistinguishable. Furthermore, the values opened during the multiplication are uniformly random values in the real world, as is the case with the simulation. At the end of the computation, parties run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ on all the values that were opened. In the real world $\mathcal{A}$ is able to cheat with probability at most $2/p$. The check is the one as the one proved in [DKL+13], so we refer the reader to it for a detailed analysis of $\Pi_{\mathsf{SPDZ\text{-}MAC}}$. As shown in Lemma 4.3, the probability that $\mathcal{A}$ cheats in the calls to $\mathcal{F}_{\mathsf{Prep}}$ and passes the check is $2/p$. Therefore, the overall probability of $\mathcal{A}$ cheating is negligible in p. □

Complexity Analysis.

Compared with the standard SPDZ online phase [DKL+13], our dynamic variant is more expensive, since we need to verify multiplications. Instead of 2 openings of $[\![\cdot]\!]$-shared values per multiplication, as in SPDZ, we need 4 openings of $[\![\cdot]\!]$-shared values, plus 2 openings of $[\cdot]$ sharings. This leads the overall online communication and the storage complexity to be around 3x that of SPDZ. However, our preprocessing protocol from Section 4.3 is vastly more efficient than any SPDZ preprocessing, since it is the only protocol that is PCG-friendly, allowing N triples to be preprocessed with communication scaling in $O(\log N)$. Furthermore, this comes with the additional flexibility of dynamically choosing the set of parties in the online phase.

Protocol Variants.

If supporting a dynamic committee for the online phase is not a requirement, we could modify our scheme by shifting the verification of multiplication triples to the preprocessing. This reduces the overhead of the online phase, and is essentially a regular SPDZ protocol run with our preprocessing. We simply authenticate all the c, c' components of the triples during the preprocessing phase, and then run a standard pairwise verification procedure [DPSZ12] to check one triple using another. This effectively moves the 4 extra openings in our online phase to the preprocessing, leading to an online phase with the same cost as SPDZ, although now the preprocessing has $O(N)$ complexity.

Of course, if the entire preprocessing committee $\mathcal{P}_{\mathsf{main}}$ does this, this introduces a lot more interaction from parties who may not have been involved in the online phase. Another option is to run this verification in the online committee $\mathcal{P}_{\mathsf{curr}}$ at the *start* of the online phase, after $\mathcal{P}_{\mathsf{curr}}$ has been elected, but possibly before the desired computation has been determined.

4.5 Fluid SPDZ

In this section, we show how to run Fluid SPDZ, which is a SPDZ-like online phase that supports fluidity. We base ourselves on the universal preprocessing from Section 4.3, where the entire set of parties, $\mathcal{P}_{\mathsf{main}}$, is involved. Later, in the online phase, we start with a subset of parties $\mathcal{P}_{\mathsf{curr}} \subset \mathcal{P}_{\mathsf{main}}$, and this committee can later evolve in a dynamic way (in contrast to Dynamic SPDZ, where the committee is fixed once the online phase begins). As discussed in Section 4.2, we assume when the committee changes at the end of an epoch, the current committee is made aware of the identity of the next committee who they hand-off their state to. We show how to leverage $\mathcal{F}_{\mathsf{Prep}}$ to achieve a *maximally fluid* online phase, where each epoch may last only one round. In our protocol, we will denote the current committee in a given epoch by $\mathcal{P}_{\mathsf{curr}}$. Before going into the main online protocol, we cover some key building blocks necessary to support fluidity, and describe how we adapt the SPDZ MAC check protocol to work in this context.

Simple Resharing.

We use a standard method for resharing an additively shared value $[x]^{\mathcal{P}_{\mathsf{curr}}}$ from committee $\mathcal{P}_{\mathsf{curr}}$ into committee $\mathcal{P}_{\mathsf{next}}$, as shown in Fig. 4.19. To reduce communication, we assume a setup where every pair of parties shares a common PRG seed. (If this is not available, note that we can still have parties in $\mathcal{P}_{\mathsf{curr}}$ sample and send the PRG seeds, which saves communication when a large batch of values is being reshared).

Resharing with MACs: the Key-Switch Procedure.

Since our protocol uses SPDZ $[\![\cdot]\!]$-sharing, simple resharing is not enough to securely transfer the state from one committee to another. We also need a way to securely reshare a value $[\![x]\!]$, while *switching* to a different MAC key, which is held by the second committee.

Our solution is to use the *key-switch protocol*, $\Pi_{\mathsf{Key\text{-}Switch}}$, shown in Fig. 4.20. This securely transfers $[\![x]\!]$ from $\mathcal{P}_{\mathsf{curr}}$ to $\mathcal{P}_{\mathsf{next}}$, while switching to the appropriate MAC key. The protocol proceeds as follows: each party $P_i \in \mathcal{P}_{\mathsf{curr}}$ starts with a random value r^i that is pairwise authenticated with every party in $\mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}$ — that is, P_i holds a MAC on t^i under P_j's MAC key, for each $P_j \in \mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}$. This can easily be obtained by a call to $\mathcal{F}_{\mathsf{Prep}}$ using the Rand command. Each P_i can then obtain $[\Delta_{\mathcal{P}_{\mathsf{curr}}} \cdot t]$, where $t = \sum_{i \in \mathcal{P}_{\mathsf{curr}}} t^i$, by combining the relevant MAC shares as in Π_{Convert}, thus forming $[\![t]\!]$. The idea now is for $\mathcal{P}_{\mathsf{curr}}$ to open the masked value $x + t$, which $\mathcal{P}_{\mathsf{next}}$ can use to obtain $[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot x] = [\Delta_{\mathcal{P}_{\mathsf{next}}}] \cdot (x + t) - [\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t]$. All that remains is for parties in $\mathcal{P}_{\mathsf{next}}$ to get $[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t]$. Note that $\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t = \sum_{i \in \mathcal{P}_{\mathsf{curr}}} \sum_{j \in \mathcal{P}_{\mathsf{next}}} M_j^i - K_i^j$. Therefore, the parties in $\mathcal{P}_{\mathsf{curr}}$ can reshare $M = \sum_{j \in \mathcal{P}_{\mathsf{next}}} M_j^i$ to parties in $\mathcal{P}_{\mathsf{next}}$, who then locally sum the shares and their keys to obtain shares of $\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t = M - \sum_{i \in \mathcal{P}_{\mathsf{curr}}} K_i^j$. Security of $\Pi_{\mathsf{Key\text{-}Switch}}$ is stated and analysed in Lemma 4.4.

Lemma 4.4. *If parties in $\mathcal{P}_{\mathsf{curr}}$ follow the protocol, $\Pi_{\mathsf{Key\text{-}Switch}}$ leads to a consistent sharing of $[\![x]\!]^{\mathcal{P}_{\mathsf{curr}}}$, and its transcript is simulatable by random values.*

Proof. Consider a committee $\mathcal{P}_{\mathsf{curr}}$ running $\Pi_{\mathsf{Key\text{-}Switch}}$ on a $[\![\cdot]\!]$-shared value x. They begin by calling $\mathcal{F}_{\mathsf{Prep}}$ to receive a $\langle\cdot\rangle$-shared random t. $\mathcal{P}_{\mathsf{curr}}$ then locally applies Π_{Convert} to get $[\![t]\!]$. Note that,

$$\begin{aligned}
&M_j^i = \Delta^j \cdot t + K_i^j, \forall j \in \mathcal{P}_{\mathsf{next}},\\
&\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t = \sum_{i \in \mathcal{P}_{\mathsf{curr}}} \sum_{j \in \mathcal{P}_{\mathsf{next}}} M_j^i - K_i^j\\
&(\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t)^j = [M] - [K], \text{ where } M = \sum_{j \in \mathcal{P}_{\mathsf{next}}} M_j^i, K = \sum_{i \in \mathcal{P}_{\mathsf{curr}}} K_i^j
\end{aligned}$$

Each $P_i \in \mathcal{P}_{\mathsf{curr}}$ can compute a share of M by adding all the MACs it has with parties in $\mathcal{P}_{\mathsf{next}}$. Therefore, by resharing $[M]$, $\mathcal{P}_{\mathsf{next}}$ can compute $[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t]$. In parallel, $\mathcal{P}_{\mathsf{curr}}$ opens $[\![x + t]\!]$ to $\mathcal{P}_{\mathsf{next}}$, which $\mathcal{P}_{\mathsf{next}}$ uses to compute MAC shares on x under the key $\Delta_{\mathcal{P}_{\mathsf{next}}}$. This is still secure as the adversary does not know t in the clear so $x + t$ is uniformly random. Finally, $\mathcal{P}_{\mathsf{curr}}$ reshares $[x]$ to $\mathcal{P}_{\mathsf{next}}$.

An adversary could cheat in the opening of $[\![x + t]\!]$ or during the resharing of $[M]$ and $[x]$. In the first scenario, since we are opening an authenticated sharing, if the adversary cheats by injecting an additive error, it will get caught in the $\Pi_{\mathsf{Fluid\text{-}MAC}}$ that is run as part of Π_{Open} except with probability $2/p$.

Let the additive error by the adversary during the resharing of $[M]$ be ϵ_M and resharing of $[x]$ be ϵ_x. We show that if $\epsilon_M, \epsilon_x \neq 0$, it will result in an inconsistent MAC on x except with negligible probability. Observe that $\mathcal{P}_{\mathsf{next}}$ will compute,

$$\begin{aligned}
&[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t] = [M] - [K] + \epsilon_M,\\
&[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot x] = [\Delta_{\mathcal{P}_{\mathsf{next}}}] \cdot (x + t) - [\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t] + \epsilon_M\\
&[x] = [x] + \epsilon_x
\end{aligned}$$

At this point, one of two things can happen with $[\![x]\!]$. The first is, $\mathcal{P}_{\mathsf{next}}$ uses $[\![x]\!]$ to evaluate a multiplication gate. In this case, $[\![x - a]\!]$ will be opened using a triple (a, b, c) by running Π_{Open}, which runs a MAC Check so the adversary will get caught. The other thing that could happen is $[\![x]\!]$ is reconstructed as an output, where before accepting x, a MAC Check on the opened value is run. Therefore, the probability of the adversary cheating in $\Pi_{\mathsf{Key\text{-}Switch}}$ depends on guessing $\Delta_{\mathcal{P}_{\mathsf{next}}}$ to make $\epsilon_M = \Delta_{\mathcal{P}_{\mathsf{next}}} \cdot \epsilon_x$ to cheat in the MAC Check. Since the MAC Check has a probability of $2/p$ of failing, we conclude that the adversary gets caught in $\Pi_{\mathsf{Key\text{-}Switch}}$ except with negligible probability.

□

Fluid MAC Check:

The MAC check protocol from SPDZ (Fig. 4.16) is designed to check a large batch of MACs at the end of the computation. The protocol involves computing an additively shared $[\sigma]$, which is derived from a random linear combination of all the opened values and the corresponding MACs. We call σ the *MAC check state.* If there was no cheating, σ, when opened, should be zero. In the fluid setting, however, deferring the MAC check means that parties need to keep track of all the opened values and MACs by resharing them across committees, which blows up the complexity of the protocol. An alternative would be to run the full MAC Check protocol on values as soon as they are opened over the course of the computation. Instantiating this in a maximally fluid way would run over 4 epochs. Instead, we propose an incremental version of the check that updates the MAC check state in every epoch, using a fresh random challenge to serve as the next linear combination coefficient. This essentially compresses the number of things to be checked down to a constant size. Another advantage of the incremental check is that it only runs over 2 epochs.

$\Pi_{\mathsf{Fluid\text{-}MAC}}$, detailed in Fig. 4.21, has two subprotocols. During the online computation, parties run **Compute State** to incrementally update the MAC check state, the shared value $[\sigma]$ (which is initially zero). At the end of the computation, the final committee runs **Check State** to check that the $[\sigma]$ is still zero. Let $(A_1, \ldots, A_m)$ be a set of opened values that $\mathcal{P}_i$ wants to check the MACs on. We assume that $\mathcal{P}_{i+1}$ holds the shared state $[\sigma']$, from prior epochs. The protocol begins with $\mathcal{P}_i$, which opens a random challenge β from $\mathcal{F}_{\mathsf{Prep}}$ to $\mathcal{P}_{i+1}$; since β is obtained in $\langle\cdot\rangle$ form, $\mathcal{P}_{i+1}$ can locally check the MACs on β to verify this. By taking a linear combination with powers of β, $\mathcal{P}_{i+1}$ computes $[\sigma] = [\sigma'] + \gamma^k - [\Delta_{\mathcal{P}_i}] \cdot A$, where $A = \sum_{j=1}^{m} \beta^j \cdot A_j$ and $\gamma^k = \sum_{j=1}^{m} \beta^j \cdot [\Delta_{\mathcal{P}_i} \cdot A_j]$.

At the end of the protocol, when a committee wants to complete the MAC Check, all it has to do is securely open $[\sigma]$ and check that it is zero.

Fluid Verify:

In $\Pi_{\mathsf{Fluid\text{-}Verify}}$, parties in a given committee, say $\mathcal{P}_{i+1}$, want to verify the outputs of multiplication gates using the randomised circuit outputs, similar to the verification method from Section 4.4. As in the Fluid MAC check, we carry out the check incrementally throughout the computation, where in the first phase, the parties open a random value, which is expanded into challenges $\alpha_i \in \mathbb{F}_p$, used to update the sharings $[\![u]\!], [\![w]\!]$, corresponding to the tally of randomised multiplications and actual multiplications. These are maintained as state, until the final verification phase where we open $[\![r]\!]$ and check that $[\![u]\!] - r \cdot [\![w]\!] = 0$. The underlying technique is similar to the one used in [CGG+21], and the protocol appears in Fig. 4.22.

Fluid Online:

We now describe how the online phase works. $\Pi_{\mathsf{Fluid\text{-}Online}}$ begins the same way as $\Pi_{\mathsf{SPDZ\text{-}Online}}$ with a set of parties $\mathcal{P}_{\mathsf{curr}} \subseteq \mathcal{P}_{\mathsf{main}}$, running Input and Initialise phases. These are used to set up the preprocessing functionality, and create authenticated sharings of the inputs. During these two phases, we assume that the committee does not change. Addition and multiplication by a public constant are local operations, so they are naturally maximally fluid operations.

Multiplication needs to be spread out over multiple epochs to do it in a maximally fluid way. To evaluate one multiplication between x, y, we need to perform two multiplications: $x \cdot y$ and $rx \cdot y$. At a high level, we can think of parties doing two things in $\Pi_{\mathsf{Fluid\text{-}Mult}}$. The first is computing output shares of the multiplications $[\![z]\!], [\![rz]\!]$. The second thing is running the MAC check and the verification protocols in an incremental way, so that we retain a small state complexity throughout the computation. Both of these parts are run in parallel between the committees $\mathcal{P}_{\mathsf{curr}-1}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}+1}$.

The full online phase is given in Fig. 4.23. Below, we focus on describing the multiplication protocol, shown in Fig. 4.24.

Computing the output shares. In order for the current committee $\mathcal{P}_{\mathsf{curr}}$ to evaluate the multiplications, we start with the committee of the previous epoch $\mathcal{P}_{\mathsf{curr\text{-}1}}$. We want to use $\mathcal{P}_{\mathsf{curr\text{-}1}}$ to set up an authenticated triple for $\mathcal{P}_{\mathsf{curr}}$ to use. Towards this, $\mathcal{P}_{\mathsf{curr\text{-}1}}$ calls $\mathcal{F}_{\mathsf{Prep}}$ to receive two triples - $(\langle a \rangle, \langle b \rangle, [c])$ and $(\langle a' \rangle, \langle b' \rangle, [c'])$. In addition, they also call it using Rand to receive authenticated shares of two random values $\langle l \rangle$ and $\langle l' \rangle$, to be used to authenticate $[c], [c']$. Parties use Π_{Convert} to locally go from $\langle\cdot\rangle$ to $[\![\cdot]\!]$ shares of the triples and the random values. To transfer the triples to $\mathcal{P}_{\mathsf{curr}}$ such that the MACs are under their key, $\mathcal{P}_{\mathsf{curr\text{-}1}}$ runs the $\Pi_{\mathsf{Key\text{-}Switch}}$ protocol with $\mathcal{P}_{\mathsf{curr}}$, on $([\![a]\!], [\![b]\!]), ([\![a']\!], [\![b']\!]), [\![l]\!], [\![l']\!]$ and opens $[l + c], [l' + c']$ to them. As a result, $\mathcal{P}_{\mathsf{curr}}$ can locally get authenticated shares of the triples under the MAC key $\Delta_{\mathcal{P}_{\mathsf{curr}}}$. Using shares of the triples, they locally compute $[\![x - a]\!], [\![y - b]\!], [\![x - a']\!], [\![y - b']\!]$ and open them to $\mathcal{P}_{\mathsf{curr}+1}$. $\mathcal{P}_{\mathsf{curr}+1}$ can compute $[\![z]\!], [\![rz]\!]$ using the standard Beaver multiplication technique.

Security of the Online Protocol.

We now briefly discuss security of the online protocol, $\Pi_{\mathsf{Fluid\text{-}Online}}$. As argued in Lemma 4.4, the values sent in the key-switch protocol are always indistinguishable from random, and any errors in the resulting sharing will always be detected by a MAC check. Regarding $\Pi_{\mathsf{Fluid\text{-}MAC}}$ and $\Pi_{\mathsf{Fluid\text{-}Verify}}$, note that these protocols both follow essentially the same set of steps as the Dynamic SPDZ protocols ($\Pi_{\mathsf{SPDZ\text{-}MAC}}$ and $\Pi_{\mathsf{SPDZ\text{-}Verify}}$). The key differences are (1) the random challenges are obtained by opening random authenticated sharings, instead of $\mathcal{F}_{\mathsf{Coin}}$, and (2) the final check values are computed incrementally, instead of immediately. For (1), because the sharings are authenticated and MACs immediately checked, they are still uniformly random until the time of opening. For (2), note that since each challenge is only opened after the corresponding value being checked has been made public, its randomness still contributes in the same way as Dynamic SPDZ, to prevent cheating.

During the multiplication protocol, $\Pi_{\mathsf{Fluid\text{-}Mult}}$, the parties run the same computations as in Dynamic SPDZ, with the difference that in each round, the state is securely transferred using Π_{Reshare} or $\Pi_{\mathsf{Key\text{-}Switch}}$, and the MAC check and verification procedures are run in the background. Hence, security can be proven similarly to the proof of Theorem 4.4. We obtain the following.

Theorem 4.5. *Let $\mathcal{A}$ be an* R*-adaptive adversary in $\Pi_{\mathsf{Fluid\text{-}Online}}$. Then, the protocol UC-securely computes $\mathcal{F}_{\mathsf{DABB}}$ in the presence of $\mathcal{A}$ in the $\mathcal{F}_{\mathsf{Prep}}$-hybrid model.*

Protocol Variants:

Similar to the variants considered for the Dynamic SPDZ protocol, we can shift some of the costs involved in $\Pi_{\mathsf{Fluid\text{-}Online}}$ to a post-preprocessing phase. We can make the model slightly more restrictive by having the parties communicate the epochs of the online phase in which they would be active, at the end of the preprocessing phase. The committees are now known, which means parties can communicate within their committees to authenticate triples before the function to be computed is determined. Since the triples are authenticated by the time the online computation starts, we do not need $\mathcal{P}_{\mathsf{curr}-1}$ to send the triple to $\mathcal{P}_{\mathsf{curr}}$, saving in terms of communication.

4.6 Cost Analysis

In Table 4.1 we give some efficiency estimates for our protocols, in terms of the per-party communication and storage costs. n is the number of parties, while n_c is the average committee size in the online phase. First, in the preprocessing, our dynamic and fluid protocols have significantly smaller storage and communication compared with previous SPDZ protocols (if n is small, relative to the circuit size). As mentioned in Section 4.4, we can also use our preprocessing to get a modified version of SPDZ, with the same online cost as regular SPDZ, by verifying the multiplication triples in the offline phase. This gives the best preprocessing complexity for any SPDZ-like protocol with the same online phase.

Table 4.1: Cost estimates for various protocols (comm. in # field elements per party)

Protocol	Online comm.	Preproc. comm.	Storage
SPDZ [KPR18, KOS16]	$2n\|C\|$	$O(n\|C\|)$	$O(\|C\|)$
[BGIN22]	$2n\|C\|$	$O(n\sqrt{\|C\|})$	$O(\sqrt{\|C\|})$
SPDZ (with our preproc.)	$2n\|C\|$	$O(\|C\|) + O(n\log(\|C\|)$	$O(\|C\|) + O(n\log(\|C\|)$
Dynamic SPDZ	$6n\|C\|$	$O(n\log(\|C\|)$	$O(n\log(\|C\|)$
Fluid SPDZ	$O(n_c\|C\|)$	$O(n\log(\|C\|)$	$O(n\log(\|C\|)$

In all protocols apart from Fluid SPDZ, the online complexities can be reduced from $O(n)$ field elements per multiplication to $O(1)$, by using the "king" approach to open values [DN07], where parties send their shares to a designated party, who sums them up and sends back the result. Although this takes an additional round, the "king" approach brings down the cost of traditional SPDZ protocols to $4|C|$ field elements per party, and $12|C|$ for Dynamic SPDZ. It seems that the king approach cannot be used to improve Fluid SPDZ, because of the need to reshare values from one committee to another.

In Table 4.1 we present asymptotic estimates of the cost of variants of our protocols against the current best SPDZ protocols [KPR18, KOS16]. The primary improvement comes from our preprocessing, which can be used to run a traditional SPDZ online phase without any fluidity, at the same cost as the other approaches. It has an additional factor of $O(|C|)$ in the preprocessing compared to Dynamic and Fluid SPDZ because we also authenticate and check the triples in the preprocessing. Comparing Dynamic SPDZ with [KPR18, KOS16] shows that we can support dynamic participants at the cost of a small overhead in the online phase, and a vastly more cheaper preprocessing phase, making it practically efficient. Compared with the recent work of [BGIN22], our preprocessing scales asymptotically better with the circuit size, although its storage costs scale worse with the number of parties, and our online phase is slightly less efficient.

To get an idea of the concrete efficiency of our universal preprocessing, we give some communication estimates based on existing VOLE and OLE protocols. For producing $N = 2^{20}$ triples, each pair of the n parties needs a VOLE of length $4N$ and an OLE of length N field elements. Using state-of-the-art LPN-based VOLE [WYKW21b] and OLE [BCG$^+$20], this can be done with a total of around 4MB of communication per pair of parties. For example, using Dynamic SPDZ with 10 parties, each party can use under 40MB of bandwidth, to gain the ability to do MPC with any subset of parties later on.

Concrete Costs and Optimizations for $\Pi_{\mathsf{Fluid\text{-}Online}}$

In this section, we estimate the concrete communication cost per party running $\Pi_{\mathsf{Fluid\text{-}Online}}$. Note that running the online phase in a maximally fluid way, as described in Fig. 4.24, allows for multiplications to be interleaved across committees. This means that parties in a committee, say $\mathcal{P}_i$, may be involved in three multiplications in parallel. This can be seen as running three instances of $\Pi_{\mathsf{Fluid\text{-}Online}}$ in parallel, with $\mathcal{P}_i$ playing different roles ($\mathcal{P}_{\mathsf{curr}\text{-}1}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}+1}$) across the three instances in parallel. In addition, we can reduce the number of random challenges that need to be opened as part of **Compute State** and **Incremental Verification** due to the interleaving.

To calculate the concrete cost, we assume that the circuit has a uniform width of m, and the committees are of size n_c. The number of elements per party per epoch can then be estimated by the following formula: $14 \cdot m \cdot n_c + 42 \cdot m + 13 \cdot n_c + 20$. If the circuit is wide, i.e. $m \gg n_c$, the amortised cost per multiplication becomes $14 \cdot n_c + 42$. The cost of adding an additional party to the computation will roughly be 14 elements.

Though we presented maximally fluid protocols, in practice one could relax the model by allowing each epoch to last more than one round. The motivation to do so is to save in terms of the concrete communication cost. For instance, assume that the fluidity is four rounds instead of one. As the multiplication in $\Pi_{\mathsf{Fluid\text{-}Online}}$ takes three rounds (including computing **Compute State** and **Incremental Verification**), this means the committee that starts the multiplication will be the one to finish it as well. There will not be a need for state transfer during the multiplication, essentially getting rid of all the Key-Switch operations in $\Pi_{\mathsf{Fluid\text{-}Online}}$. Transferring the state after the multiplication is also cheaper, as the committee will only have to Key-Switch output wires of the multiplication, the MAC key, and the random value $[\![r]\!]$. The cost of running the Fluid online with a fluidity of four is $12 \cdot m + 4 \cdot n_c$, where $12 \cdot m$ is the cost for authenticating $2m$ triples and opening the Beaver triple intermediate values, and the $4 \cdot n_c$ is for the random challenges that need to be opened for **Compute State** and **Incremental Verification**. With a wide enough circuit, the amortised cost per multiplication per party comes down to about 12 elements, matching the cost of Dynamic SPDZ.

Protocol $\Pi_{\mathsf{SPDZ\text{-}Online}}$

Init: Each $P_i \in \mathcal{P}_{\mathsf{main}}$ sends $(\mathsf{Init}, m_T, m_R)$ to $\mathcal{F}_{\mathsf{Prep}}$ and receives Δ^i. Later, when $\mathcal{P}_{\mathsf{curr}} \subseteq \mathcal{P}_{\mathsf{main}}$ wants to run the online phase, each $P_i \in \mathcal{P}_{\mathsf{curr}}$ sets $\mathsf{count} = 0, \mathsf{rcount} = 0$, and calls $\mathcal{F}_{\mathsf{Prep}}$ with $(\mathsf{Rand}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}}, \mathsf{rcount})$ to obtain $[\![r]\!]$.
Input: To share an input x, P_i inputs $(\mathsf{Rand}, P_i, \mathcal{P}_{\mathsf{curr}}, \mathsf{rcount})$ to $\mathcal{F}_{\mathsf{Prep}}$ to get $\langle t \rangle$, where P_i knows t. Then,

1. P_i samples shares of x such that $x = \sum_{j \in \mathcal{P}_{\mathsf{curr}}} x^j$ and sends $(x^j, x+t)$ to each $P_j \in \mathcal{P}_{\mathsf{curr}}$. P_i sets its share $(\Delta \cdot x)^i = \Delta^i \cdot (x+t) - (\Delta t)^i$, where $(\Delta t)^i = \Delta^i \cdot t - \sum_{j \in \mathcal{P}_{\mathsf{curr}} \setminus \{P_i\}} M_j^i$.
2. Each $P_j \in \mathcal{P}_{\mathsf{curr}} \setminus \{P_i\}$ sets its share to be $[\![x]\!] = (x^j, \Delta^j \cdot (x+t) - (\Delta t)^j)$, where $(\Delta t)^j = K_i^j$.
3. Each $P_i \in \mathcal{P}_{\mathsf{curr}}$ runs **Multiplication** below on $[\![x]\!]$ and $[\![r]\!]$ to get $[\![r \cdot x]\!]$.[a]

Addition: To perform addition, $[\![z]\!] = [\![x]\!] + [\![y]\!]$, each $P_i \in \mathcal{P}_{\mathsf{curr}}$ locally adds their shares of $[\![x]\!], [\![y]\!]$, and $[\![rx]\!], [\![ry]\!]$ to get $[\![x+y]\!], [\![r(x+y)]\!]$.
Addition by Constant: To compute $[\![z]\!] = [\![x+c]\!]$, a designated party (say P_j) adds c to its share x^j, and all parties add $\Delta^i c$ to their MAC share.
Multiplication by Constant: To compute $[\![z]\!] = k \cdot [\![x]\!]$, each $P_i \in \mathcal{P}_{\mathsf{curr}}$ locally multiply the public constant k to shares of $[\![x]\!]$ to get $[\![kx]\!], [\![r \cdot (kx)]\!]$.
Multiplication: To compute $[\![z]\!] = [\![x]\!] \cdot [\![y]\!]$ and $[\![rz]\!] = [\![rx]\!] \cdot [\![y]\!]$, each $P_i \in \mathcal{P}_{\mathsf{curr}}$:

1. Calls $\mathcal{F}_{\mathsf{Prep}}$ twice with inputs $(\mathsf{Trip}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}}, \mathsf{count})$, incrementing count after each call. $\mathcal{F}_{\mathsf{Prep}}$ outputs shares of the triples $(\langle a \rangle, \langle b \rangle, [c]), (\langle a' \rangle, \langle b' \rangle, [c'])$.
2. Calls $\mathcal{F}_{\mathsf{Prep}}$ with $(\mathsf{Rand}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}}, \mathsf{rcount})$ twice to receive $\langle l \rangle, \langle l' \rangle$. Increment rcount after each call.
3. Applies Π_{Convert} on $(\langle a \rangle, \langle b \rangle, \langle a' \rangle, \langle b' \rangle, \langle l \rangle, \langle l' \rangle)$ to get $[\![\cdot]\!]$ shares.
4. Runs Π_{Open} on $[l+c], [l'+c']$.
5. Runs Π_{Open} on $[e] = [x-a], [d] = [y-b], [e'] = [rx-a']$ and $[d'] = [y-b']$) and computes the multiplications as:

$$
\begin{aligned}
&[\Delta \cdot c] = (l+c) \cdot \Delta^j - [\Delta \cdot l], \quad [\Delta \cdot c'] = (l'+c') \cdot \Delta^j - [\Delta \cdot l] \\
&[\![z]\!] = e \cdot d + e \cdot [\![b]\!] + d \cdot [\![a]\!] + [\![c]\!] \\
&[\![rz]\!] = e' \cdot d' + e' \cdot [\![b']\!] + d' \cdot [\![a']\!] + [\![c']\!]
\end{aligned}
$$

Reconstruction: First, run $\Pi_{\mathsf{SPDZ\text{-}Verify}}$ to check the multiplications. Then, to output $[\![z]\!]$, run Π_{Open} on $[z]$, then use $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check its MAC.

[a]We actually only use one triple to multiply x and r, skipping the extra product in the protocol.

Figure 4.17: Protocol for the online phase of Dynamic SPDZ

Protocol $\Pi_{\mathsf{SPDZ\text{-}Verify}}$

Verification: Let $\{v_i, rv_i\}_{i\in[M]}$ be the input wires of the circuit, and $\{z_i, rz_i\}_{i\in[N]}$ be the output wires of multiplication gates of the circuit.

1. Parties start by running $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ to check MACs on all the values opened in multiplications and inputs previously. If $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ fails, abort, else continue.
2. Parties call $\mathcal{F}_{\mathsf{Coin}}$ to receive $\alpha_1, \ldots, \alpha_N$, $\beta_1, \ldots, \beta_M \in \mathbb{F}_p$
3. Parties locally compute

$$[\![u]\!] = \sum_{i=1}^{N} \alpha_i \cdot [\![rz_i]\!] + \sum_{i=1}^{M} \beta_i \cdot [\![rv_i]\!]$$

$$[\![w]\!] = \sum_{i=1}^{N} \alpha_i \cdot [\![z_i]\!] + \sum_{i=1}^{M} \beta_i \cdot [\![v_i]\!]$$

4. Parties open $[\![r]\!]$ by broadcasting shares of $[r]$ and running $\Pi_{\mathsf{SPDZ\text{-}MAC}}$ on it.
5. Parties locally compute $[\![u]\!] - r[\![w]\!]$, open it and run $\Pi_{\mathsf{SPDZ\text{-}MAC}}$. If the MAC check passes and $u - rw = 0$, parties Accept it and go to reconstruction, else Reject.

Figure 4.18: Protocol for the verification phase in Dynamic SPDZ

Protocol Π_{Reshare}

Setup: Each pair of parties $P_i, P_j \in \mathcal{P}_{\mathsf{main}}$ has a common PRG seed $s^{i,j}$.
Usage: $\mathcal{P}_{\mathsf{curr}}$ reshares $[x]^{\mathcal{P}_{\mathsf{curr}}}$ to $\mathcal{P}_{\mathsf{next}}$. Parties in $\mathcal{P}_{\mathsf{next}}$ are indexed from 1 to m.

1. Each $P_i \in \mathcal{P}_{\mathsf{curr}}$ computes $x^{i,j} \in \mathbb{F}_p$ as a fresh output of a PRG applied to $s^{i,j}$, for $j = 2, \ldots, m$. P_i defines $x^{i,1} = x^i - \sum_{j=2}^{m} x^{i,j}$.
2. Each P_i sends $x^{i,1}$ to P_1 in $\mathcal{P}_{\mathsf{next}}$. Each $P_j \in \mathcal{P}_{\mathsf{next}}$ defines its share as $x^j = \sum_{i\in\mathcal{P}_{\mathsf{curr}}} x^{i,j}$ (where if $j \neq 1$, $x^{i,j}$ is computed from the PRG).

Figure 4.19: Protocol for resharing values across committees

Protocol $\Pi_{\mathsf{Key\text{-}Switch}}$

Input: $[\![x]\!] = ([x], [\Delta_{\mathcal{P}_{\mathsf{curr}}} \cdot x])$ in $\mathcal{P}_{\mathsf{curr}}$.
Output: $[\![x]\!] = ([x], [\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot x])$ in $\mathcal{P}_{\mathsf{next}}$.

1. Each $P_i \in \mathcal{P}_{\mathsf{curr}}$ calls $\mathcal{F}_{\mathsf{Prep}}$ with $(\mathsf{Rand}, \mathcal{P}_{\mathsf{curr}}, \mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}, \mathsf{rcount})$ to receive $t^i, \{M_j^i\}_{j \in \mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}}$, while $P_j \in \mathcal{P}_{\mathsf{curr}} \cup \mathcal{P}_{\mathsf{next}}$ receives K_i^j.
2. $\mathcal{P}_{\mathsf{curr}}$ uses Π_{Convert} to form $[\![t]\!]^{\mathcal{P}_{\mathsf{curr}}}$. Each $P_i \in \mathcal{P}_{\mathsf{curr}}$ computes $M^i = \sum_{j \in \mathcal{P}_{\mathsf{next}}} M_j^i$ to obtain $[M]$.
3. Parties in $\mathcal{P}_{\mathsf{curr}}$ run $\Pi_{\mathsf{Open}}([\![x+t]\!])$ and $\Pi_{\mathsf{Reshare}}([M], [x])$, all to $\mathcal{P}_{\mathsf{next}}$.
4. Each $P_j \in \mathcal{P}_{\mathsf{next}}$ computes $K^j = \sum_{i \in \mathcal{P}_{\mathsf{curr}}} K_i^j$ to obtain $[K]$, and then defines $[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t] = [M] - [K]$
5. Finally, P_j can compute its share of the MAC $[\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot x]$ as $[\Delta_{\mathcal{P}_{\mathsf{next}}}] \cdot (x+t) - [\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot t]$. $\mathcal{P}_{\mathsf{next}}$ outputs $[x], [\Delta_{\mathcal{P}_{\mathsf{next}}} \cdot x]$.

Figure 4.20: Protocol to switch MAC keys

Protocol $\Pi_{\textsf{Fluid-MAC}}$

Usage: Parties in $\mathcal{P}_i$ want to check the MACs values $(A_1, \ldots, A_m)$ opened to them. We assume $\mathcal{P}_{i+1}$ gets the MAC state $[\sigma']$ from a previous run of $\Pi_{\textsf{Fluid-MAC}}$.

Compute State: Compute the MAC check state $[\sigma]$:

Committee i:

1. Each $P_j \in \mathcal{P}_i$ calls $\mathcal{F}_{\textsf{Prep}}$ with input $(\textsf{Rand}, \mathcal{P}_i, \mathcal{P}_{i+1}, \textsf{rcount})$ to receive $\langle \beta^j \rangle$.
2. Hand-off: Send β^j, M_k^j to each $P_k \in \mathcal{P}_{i+1}$, along with $A_1, \ldots, A_m$. Reshare $[\sigma'], [\Delta_{\mathcal{P}_i}], [\Delta_{\mathcal{P}_i} \cdot A_1], \ldots, [\Delta_{\mathcal{P}_i} \cdot A_m]$.

Committee $i + 1$:

3. P_k locally checks $M_k^j = \beta^j \cdot \Delta^k + K_j^k$ for all $j \in \mathcal{P}_i$, and aborts if any of them fail. Let $\beta = \sum_{j \in \mathcal{P}_i} \beta^j$.
4. It updates $[\sigma']$ as $[\sigma] = [\sigma'] + \gamma^k - [\Delta_{\mathcal{P}_i}] \cdot A$, where $A = \sum_{j=1}^m (\beta)^j \cdot A_j$ and $\gamma^k = \sum_{j=1}^m (\beta)^j \cdot [\Delta_{\mathcal{P}_i} \cdot A_j]$ (here, $(\beta)^j$ is the j-th power of β).

Check State: (**Committee $i + 2$**)

5. Set $\sigma^j = \sum_{k \in \mathcal{P}_{i+1}} [\sigma^k]$. Each $P_j \in \mathcal{P}_{i+2}$ calls $\mathcal{F}_{\textsf{Commit}}$ to commit to σ^j.
6. Open all commitments, and if they are consistent, Accept if $\sum_{j \in \mathcal{P}_{i+2}} \sigma^j = 0$. Else, Reject.

Figure 4.21: MAC check protocol for a fluid committee

Protocol $\Pi_{\mathsf{Fluid\text{-}Verify}}$

Usage: Parties in $\mathcal{P}_{i+1}$ want to verify the output wires of multiplication gates of layer l, denoted by $\{z_j, rz_j\}_{j=1}^{N}$. We assume that $\mathcal{P}_{i+1}$ gets the state $[\![u']\!], [\![w']\!]$ from a previous run of $\Pi_{\mathsf{Fluid\text{-}Verify}}$.

Incremental Verification:

Committee i:

1. Each $P_j \in \mathcal{P}_i$ calls $\mathcal{F}_{\mathsf{Prep}}$ with $(\mathsf{Rand}, \mathcal{P}_i, \mathcal{P}_{i+1}, \mathsf{rcount})$ to receive $\langle s \rangle$.
2. Hand-off: P_j sends the share s^j and MAC M_k^j to each $P_k \in \mathcal{P}_{i+1}$, and runs $\Pi_{\mathsf{Key\text{-}Switch}}$ on $[\![u']\!], [\![w']\!]$.

Committee $i+1$:

3. P_k locally checks $M_k^j = s^j \cdot \Delta^k + K_j^k$ for all $j \in \mathcal{P}_i$, and aborts if any fail. Let $s = \sum_{j \in \mathcal{P}_i} s^j$. Using s as a seed for PRG, generate pseudorandom $\alpha_1, \ldots, \alpha_N \in \mathbb{F}_p$.
4. Each P_k locally computes $[\![u]\!] = [\![u']\!] + \sum_{i=1}^{N} \alpha_i \cdot [\![rz_i]\!]$ and $[\![w]\!] = [\![w']\!] + \sum_{i=1}^{N} \alpha_i \cdot [\![z_i]\!]$.

Final Check:

Committee $i+2$:

5. Parties in $\mathcal{P}_{i+2}$ start by running $\Pi_{\mathsf{Key\text{-}Switch}}$ with $\mathcal{P}_{i+1}$ to receive $[\![u]\!], [\![w]\!]$ under $\Delta_{\mathcal{P}_{i+2}}$.
6. Then they run the **Check MACs** phase of $\Pi_{\mathsf{Fluid\text{-}MAC}}$. If $\Pi_{\mathsf{Fluid\text{-}MAC}}$ fails, Reject, else continue.
7. They execute Π_{Open} on $[\![r]\!]$ to receive r, and check its MAC with $\Pi_{\mathsf{Fluid\text{-}MAC}}$.
8. Parties compute $\Pi_{\mathsf{Open}}([\![u]\!] - r[\![w]\!])$, then check the MAC. If the opened value is 0, parties Accept and go to reconstruction, else Reject.

Figure 4.22: Verification phase for a fluid computation

Protocol $\Pi_{\textsf{Fluid-Online}}$

Init: Every $P_i \in \mathcal{P}_{\textsf{curr}} \subseteq \mathcal{P}_{\textsf{main}}$ sets $\textsf{count} = 0, \textsf{rcount} = 0$. P_i inputs $(\textsf{Rand}, \mathcal{P}_{\textsf{curr}}, \mathcal{P}_{\textsf{curr}}, \textsf{rcount})$ to $\mathcal{F}_{\textsf{Prep}}$ and receives $\langle r \rangle$. P_i sends $(\textsf{Init}, m_T, m_R)$ to $\mathcal{F}_{\textsf{Prep}}$ and receives Δ^i.

Input: To form $[\![\cdot]\!]$-sharing of an input x possessed by $P_i \in \mathcal{P}_{\textsf{main}}$,

1. P_i along with parties in $\mathcal{P}_{\textsf{curr}}$ runs $\Pi_{\textsf{Key-Switch}}$, where P_i (acting as $\mathcal{P}_{\textsf{curr}}$) inputs $[\![x]\!]$ under its key and parties in $\mathcal{P}_{\textsf{curr}}$ (as $\mathcal{P}_{\textsf{next}}$) receive $[\![x]\!]$ under their key.

2. Parties in $\mathcal{P}_{\textsf{curr}}$ input $(\textsf{Trip}, \mathcal{P}_{\textsf{curr}}, \mathcal{P}_{\textsf{curr}}, \textsf{count})$ to $\mathcal{F}_{\textsf{Prep}}$ and receive $(\langle a \rangle, \langle b \rangle, [c])$.

3. Then they engage to perform the multiplication of $\{[\![x_i]\!]\}_{i \in \mathcal{P}_{\textsf{curr}}}$ with $[\![r]\!]$ to produce $\{[\![r \cdot x_i]\!]\}_{i \in \mathcal{P}_{\textsf{curr}}}$.

Addition: To perform addition, $[\![z]\!] = [\![x]\!] + [\![y]\!]$, each $P_i \in \mathcal{P}_{\textsf{curr}}$ locally adds their shares of $[\![x]\!], [\![y]\!], [\![rx]\!], [\![ry]\!]$ to get $[\![x + y]\!], [\![r(x + y)]\!]$.
Addition by Constant: To compute $[\![z]\!] = [\![x + c]\!]$, a designated party (say $P_j \in \mathcal{P}_{\textsf{curr}}$) adds c to its share x^j, and all the other parties add $\Delta^i c$ to their MAC share.
Multiplication by Constant: To compute $[\![z]\!] = k \cdot [\![x]\!]$, each $P_i \in \mathcal{P}_{\textsf{curr}}$ locally multiply the public constant k to shares of $[\![x]\!]$ to get $[\![kx]\!], [\![r \cdot (kx)]\!]$.
Multiplication: To compute $[\![z]\!] = [\![x]\!] \cdot [\![y]\!]$ and $[\![rz]\!] = [\![rx]\!] \cdot [\![y]\!]$ in $\mathcal{P}_{\textsf{curr}}$, run $\Pi_{\textsf{Fluid-Mult}}$ among $(\mathcal{P}_{\textsf{curr-1}}, \mathcal{P}_{\textsf{curr}}, \mathcal{P}_{\textsf{curr+1}})$.

Verify and Reconstruct:

1. Parties in the final committee, say $\mathcal{P}_{final}$, run **Compute State** of $\Pi_{\textsf{Fluid-MAC}}$. If $\Pi_{\textsf{Fluid-MAC}}$ fails, Reject, else continue.

2. Parties execute **Final Check** phase of $\Pi_{\textsf{Fluid-Verify}}$. If the result is Accept, for each output wire z, they open $[\![z]\!]$ by broadcasting their shares to the other parties and running both phases of $\Pi_{\textsf{Fluid-MAC}}$. If $\Pi_{\textsf{Fluid-MAC}}$ fails, Reject.

Figure 4.23: Protocol for a maximally fluid online phase

Protocol $\Pi_{\mathsf{Fluid\text{-}Mult}}$

Usage: $\mathcal{P}_{\mathsf{curr}}$ wants to evaluate multiplications $z = x \cdot y, rz = rx \cdot y$.

Committee $\mathcal{P}_{\mathsf{curr}\text{-}1}$:

1. Calls $\mathcal{F}_{\mathsf{Prep}}$ twice with $(\mathsf{Trip}, \mathcal{P}_{\mathsf{curr}\text{-}1}, \mathcal{P}_{\mathsf{curr}\text{-}1}, \mathsf{count})$, incrementing count after each call. $\mathcal{F}_{\mathsf{Prep}}$ outputs shares of the triples $(\langle a \rangle, \langle b \rangle, [c])$, $(\langle a' \rangle, \langle b' \rangle, [c'])$.
2. Calls $\mathcal{F}_{\mathsf{Prep}}$ with $(\mathsf{Rand}, \mathcal{P}_{\mathsf{curr}\text{-}1}, \mathcal{P}_{\mathsf{curr}\text{-}1}, \mathsf{rcount})$ twice to receive $\langle l \rangle, \langle l' \rangle$, incrementing rcount after each call.
3. Applies Π_{Convert} to get on $(\langle a \rangle, \langle b \rangle, \langle a' \rangle, \langle b' \rangle, \langle l \rangle, \langle l' \rangle)$ to get $[\![\cdot]\!]$ shares. Locally computes $[l + c], [l' + c']$.
4. Hand-off:
   a) Run $\Pi_{\mathsf{Key\text{-}Switch}}$ on $([\![a]\!], [\![b]\!])$, $([\![a']\!], [\![b']\!])$, $[\![l]\!], [\![l']\!]$, and $[\![r]\!]$.
   b) Run Π_{Open} on $[l + c], [l' + c']$.

Committee $\mathcal{P}_{\mathsf{curr}}$:

5. Locally computes

$$\begin{aligned}
&[c] = (l + c) - [l], \quad [c'] = (l' + c') - [l'] \\
&[\Delta_{\mathcal{P}_{\mathsf{curr}}} \cdot c] = [\Delta_{\mathcal{P}_{\mathsf{curr}}}] \cdot (l + c) - [\Delta_{\mathcal{P}_{\mathsf{curr}}} \cdot l] \\
&[\Delta_{\mathcal{P}_{\mathsf{curr}}} \cdot c'] = [\Delta_{\mathcal{P}_{\mathsf{curr}}}] \cdot (l' + c') - [\Delta_{\mathcal{P}_{\mathsf{curr}}} \cdot l']
\end{aligned}$$

6. In addition, they also compute $[\![x - a]\!], [\![y - b]\!], [\![x - a']\!], [\![y - b']\!]$.
7. Executes Steps 1, 2 in **Incremental Verification** of $\Pi_{\mathsf{Fluid\text{-}Verify}}$ and **Compute State** in $\Pi_{\mathsf{Fluid\text{-}MAC}}$.
8. Hand-off : In parallel to the Hand-off in **Incremental Verification** and **Compute State**,
   a) Run $\Pi_{\mathsf{Key\text{-}Switch}}$ on $([\![a]\!], [\![b]\!], [\![c]\!])$, $([\![a']\!], [\![b']\!], [\![c']\!])$, $[\![r]\!]$, and $[\![\boldsymbol{m}]\!]$, where $[\![\boldsymbol{m}]\!]$ is the set of wires not used in a multiplication in the current layer.
   b) Run Π_{Open} on $[\![x - a]\!], [\![y - b]\!], [\![rx - a']\!], [\![y - b']\!]$.

Committee $\mathcal{P}_{\mathsf{curr}+1}$:

9. Locally executes the remaining steps of key-switch, and evaluates the multiplications as:

$$\begin{aligned}
&e = x - a, d = y - b, \quad e' = rx - a', d' = y - b' \\
&[\![z]\!] = e \cdot d + e \cdot [\![b]\!] + d \cdot [\![a]\!] + [\![c]\!] \\
&[\![rz]\!] = e' \cdot d' + e' \cdot [\![b']\!] + d' \cdot [\![a']\!] + [\![c']\!]
\end{aligned}$$

10. Executes Steps 3 and 4 in **Incremental Verification** of $\Pi_{\mathsf{Fluid\text{-}Verify}}$ on $[\![z]\!], [\![rz]\!]$ and in the **Compute State** phase in $\Pi_{\mathsf{Fluid\text{-}MAC}}$ on $(x - a, y - b, rx - a', y - b')$.

Figure 4.24: Protocol for a maximally fluid multiplication

Bibliography

[ABF$^+$18] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida. Generalizing the SPDZ compiler for other protocols. In *ACM CCS 2018*. ACM Press, October 2018.

[ADI$^+$17] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. Secure arithmetic computation with constant computational overhead. In *CRYPTO 2017, Part I*, LNCS. Springer, Heidelberg, August 2017.

[AFL$^+$16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *ACM CCS 2016*. ACM Press, October 2016.

[AOR$^+$19] Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. In *WAHC '19: Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. ACM, 2019. `https://eprint.iacr.org/2019/974`.

[BCG$^+$19a] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In *ACM CCS 2019*. ACM Press, November 2019.

[BCG$^+$19b] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In *CRYPTO 2019, Part III*, LNCS. Springer, Heidelberg, August 2019.

[BCG$^+$20] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators from ring-LPN. In *CRYPTO 2020, Part II*, LNCS. Springer, Heidelberg, August 2020.

[BCGI18] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In *ACM CCS 2018*. ACM Press, October 2018.

[BDK$^+$18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In *ACM CCS 2018*. ACM Press, October 2018.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT 2011*, LNCS. Springer, Heidelberg, May 2011.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO'91*, LNCS. Springer, Heidelberg, August 1992.

[BGG$^+$20] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In *TCC 2020, Part I*, LNCS. Springer, Heidelberg, November 2020.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT 2015, Part II*, LNCS. Springer, Heidelberg, April 2015.

[BGI19] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In *TCC 2019, Part I*, LNCS. Springer, Heidelberg, December 2019.

[BGIN22] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Secure multiparty computation with sublinear preprocessing. In *EUROCRYPT 2022*. Springer, Heidelberg, May 2022.

[BLN$^+$15] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015. https://eprint.iacr.org/2015/472.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS 2008*, LNCS. Springer, Heidelberg, October 2008.

[Bra85] Gabriel Bracha. An $O(\lg n)$ expected rounds randomized byzantine generals protocol. In *17th ACM STOC*. ACM Press, May 1985.

[BST20] Charlotte Bonte, Nigel P. Smart, and Titouan Tanguy. Thresholdizing hasheddsa: Mpc to the rescue. Cryptology ePrint Archive, Report 2020/214, 2020. https://eprint.iacr.org/2020/214.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, (1), January 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*. IEEE Computer Society Press, October 2001.

[Cd10] Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In *SCN 10*, LNCS. Springer, Heidelberg, September 2010.

[CDE$^+$18] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In *CRYPTO 2018, Part II*, LNCS. Springer, Heidelberg, August 2018.

[CGG$^+$21] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. Fluid MPC: Secure multiparty computation with dynamic participants. In *CRYPTO 2021, Part II*, LNCS. Springer, Heidelberg, August 2021.

[CGH$^+$18] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In *CRYPTO 2018, Part III*, LNCS. Springer, Heidelberg, August 2018.

[CGMV18] Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. ALGORAND AGREEMENT: Super fast and partition resilient byzantine agreement. Cryptology ePrint Archive, Report 2018/377, 2018. https://eprint.iacr.org/2018/377.

[CGR$^+$19] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable and efficient secure two-party computation for machine learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 496–511. IEEE, 2019.

[CL17] Ran Cohen and Yehuda Lindell. Fairness versus guaranteed output delivery in secure multiparty computation. *Journal of Cryptology*, (4), October 2017.

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *18th ACM STOC*. ACM Press, May 1986.

[Con17] Kate Conger. How apple says it prevented face id from being racist, 2017. https://gizmodo.com/how-apple-says-it-prevented-face-id-from-being-racist-1819557448. Accessed November 26, 2022.

[CS10] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In *FC 2010*, LNCS. Springer, Heidelberg, January 2010.

[DDN$^+$16] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. In *FC 2016*, LNCS. Springer, Heidelberg, February 2016.

[DEF$^+$19] Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2019.

[DEK19] Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. Cryptology ePrint Archive, Report 2019/131, 2019. https://eprint.iacr.org/2019/131.

[DEK20] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *PoPETs*, (4), October 2020.

[DEK21] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Fantastic four: Honest-majority four-party secure computation with malicious security. In *USENIX Security 2021*. USENIX Association, August 2021.

[DFK$^+$06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC 2006*, LNCS. Springer, Heidelberg, March 2006.

[DGN$^+$17] Nico Döttling, Satrajit Ghosh, Jesper Buus Nielsen, Tobias Nilges, and Roberto Trifiletti. TinyOLE: Efficient actively secure two-party computation from oblivious linear function evaluation. In *ACM CCS 2017*. ACM Press, October / November 2017.

[DKL$^+$13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS 2013*, LNCS. Springer, Heidelberg, September 2013.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO 2007*, LNCS. Springer, Heidelberg, August 2007.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO 2012*, LNCS. Springer, Heidelberg, August 2012.

[DSZ15] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015.

[EGK$^+$20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *CRYPTO 2020, Part II*, LNCS. Springer, Heidelberg, August 2020.

[FKOS15] Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In *ASIACRYPT 2015, Part I*, LNCS. Springer, Heidelberg, November / December 2015.

[FLNW17] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In *EUROCRYPT 2017, Part II*, LNCS. Springer, Heidelberg, April / May 2017.

[GHK$^+$21] Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. YOSO: You only speak once - secure MPC with stateless ephemeral roles. In *CRYPTO 2021, Part II*, LNCS. Springer, Heidelberg, August 2021.

[GIP$^+$14] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In *46th ACM STOC*. ACM Press, May / June 2014.

[GIW16] Daniel Genkin, Yuval Ishai, and Mor Weiss. Binary AMD circuits from secure multiparty computation. In *TCC 2016-B, Part I*, LNCS. Springer, Heidelberg, October / November 2016.

[GKM$^+$20] Vipul Goyal, Abhiram Kothapalli, Elisaweta Masserova, Bryan Parno, and Yifan Song. Storing and retrieving secrets on a blockchain. Cryptology ePrint Archive, Report 2020/504, 2020. `https://eprint.iacr.org/2020/504`.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *19th ACM STOC*. ACM Press, May 1987.

[GRW18] S. Dov Gordon, Samuel Ranellucci, and Xiao Wang. Secure computation with low communication from cross-checking. In *ASIACRYPT 2018, Part III*, LNCS. Springer, Heidelberg, December 2018.

[GSY21] S. Dov Gordon, Daniel Starin, and Arkady Yerukhimovich. The more the merrier: Reducing the cost of large scale MPC. In *EUROCRYPT 2021, Part II*, LNCS. Springer, Heidelberg, October 2021.

[HJKY95] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *CRYPTO'95*, LNCS. Springer, Heidelberg, August 1995.

[HKS$^+$10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. TASTY: tool for automating secure two-party computations. In *ACM CCS 2010*. ACM Press, October 2010.

[HSS17] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In *ASIACRYPT 2017, Part I*, LNCS. Springer, Heidelberg, December 2017.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *CRYPTO 2003*, LNCS. Springer, Heidelberg, August 2003.

[IMZ19] Muhammad Ishaq, Ana L. Milanova, and Vassilis Zikas. Efficient MPC via program analysis: A framework for efficient optimal mixing. In *ACM CCS 2019*. ACM Press, November 2019.

[IOZ14] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In *CRYPTO 2014, Part II*, LNCS. Springer, Heidelberg, August 2014.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. Cryptology ePrint Archive, Report 2020/521, 2020. `https://eprint.iacr.org/2020/521`.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In *ACM CCS 2016*. ACM Press, October 2016.

[KPPS21] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and robust privacy-preserving machine learning. In *USENIX Security 2021*. USENIX Association, August 2021.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT 2018, Part III*, LNCS. Springer, Heidelberg, April / May 2018.

[KPRS21] Nishat Koti, Arpita Patra, Rahul Rachuri, and Ajith Suresh. Tetrad: Actively secure 4PC for secure training and inference. Cryptology ePrint Archive, Report 2021/755, 2021. https://eprint.iacr.org/2021/755.

[Kra22] Felix Krause. iOS privacy: Announcing inappbrowser.com - see what javascript commands get injected through an in-app browser, 2022. tinyurl.com/3z968rn6. Accessed November 26, 2022.

[KS20] Marcel Keller and Ke Sun. Effectiveness of mpc-friendly softmax replacement, 2020.

[KS22] Marcel Keller and Ke Sun. Secure quantized training for deep learning. Cryptology ePrint Archive, Report 2022/933, 2022. https://eprint.iacr.org/2022/933.

[MR18] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In *ACM CCS 2018*. ACM Press, October 2018.

[MR19] Payman Mohassel and Peter Rindal. ABY3, 2019. https://github.com/ladnir/aby3/.

[MRZ15] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. In *ACM CCS 2015*. ACM Press, October 2015.

[MZ17] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2017.

[MZW$^+$19] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. CHURP: Dynamic-committee proactive secret sharing. In *ACM CCS 2019*. ACM Press, November 2019.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *CRYPTO 2012*, LNCS. Springer, Heidelberg, August 2012.

[OB$^+$19] OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning, 2019.

[RB89] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *21st ACM STOC*. ACM Press, May 1989.

[RDN$^+$22] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents, 2022.

[RS19] Rahul Rachuri and Ajith Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. Cryptology ePrint Archive, Report 2019/1315, 2019. https://eprint.iacr.org/2019/1315.

[RS21] Rahul Rachuri and Peter Scholl. Le mans: Dynamic and fluid MPC for dishonest majority. Cryptology ePrint Archive, Report 2021/1579, 2021. `https://eprint.iacr.org/2021/1579`.

[RS22] Rahul Rachuri and Peter Scholl. Le mans: Dynamic and fluid MPC for dishonest majority. In *CRYPTO 2022, Part I*, LNCS. Springer, Heidelberg, August 2022.

[RST$^+$19] Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. Actively secure setup for SPDZ. Cryptology ePrint Archive, Report 2019/1300, 2019. `https://eprint.iacr.org/2019/1300`.

[RW19] Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In *INDOCRYPT 2019*, LNCS. Springer, Heidelberg, December 2019.

[SHM$^+$16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

[SSW17] Peter Scholl, Nigel P. Smart, and Tim Wood. When it's all just too much: Outsourcing MPC-preprocessing. In *16th IMA International Conference on Cryptography and Coding*, LNCS. Springer, Heidelberg, December 2017.

[TW19] Stuart A. Thompson and Charlie Warzel. Twelve million phones, one dataset, zero privacy, 2019. `https://www.nytimes.com/interactive/2019/12/19/opinion/location-tracking-cell-phone.html`. Accessed November 26, 2022.

[WGC19] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *PoPETs*, (3), July 2019.

[WRK17a] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In *ACM CCS 2017*. ACM Press, October / November 2017.

[WRK17b] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In *ACM CCS 2017*. ACM Press, October / November 2017.

[WYKW20] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. Cryptology ePrint Archive, Report 2020/925, 2020. `https://eprint.iacr.org/2020/925`.

[WYKW21a] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 2021.

[WYKW21b] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. 42nd IEEE Symposium on Security and Privacy (Oakland 2021), 2021.

[Yao] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982).*

[ZLWL21] Zongyang Zhang, Tong Li, Zhuo Wang, and Jianwei Liu. Redactable transactions in consortium blockchain: Controlled by multi-authority CP-ABE. In *ACISP 21*, LNCS. Springer, Heidelberg, December 2021.