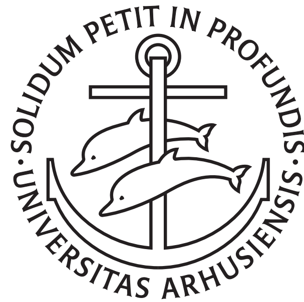
Cats or Croissants?
Techniques for Secure Inference

Anders Dalskov

PhD Dissertation



Department of Computer Science
Aarhus University
Denmark

Cats or Croissants?
Techniques for Secure Inference

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

by
Anders Dalskov
November 11, 2020

Abstract

Machine Learning is making its way into more and more aspects of our digital lives. In many applications, Machine Learning is deployed in an "outsourced" model where the device with the input (e.g., an image) is a different device, owned by a different entity, than the device with the model (e.g., a Convolutional Neural Network). Moreover, many modern applications of Machine Learning operate on sensitive data, be it the images we take or the text we write, and so being able to stay in control of this data, while still being able to benefit from Machine Learning, is important moving forward.

Secure Multiparty Computation, or MPC, presents an attractive solution to exactly this problem, and this thesis presents three concrete ways in which MPC can be used for secure Machine Learning.

The first application demonstrates protocols for secure computation with active security that enjoy particular properties that are very well suited for Machine Learning. The protocols we presented constitute concretely very efficient protocols for secure computation within the specific setting we consider: active security with an honest majority over rings. We present two instantiations: one for 3 parties that is at least as efficient as prior work, and one for any number of parties which is the first of its kind.

The second application looks at how we would obtain models that are efficient to evaluate securely. Indeed, secure inference (that is, evaluating a Machine Learning model securely on some private input) requires various tweaks with respect to the encoding of values and non-linear functions that are used. We observe that specific models that were designed to be evaluated on e.g., smartphones, also work very well for secure inference, and we present a wide array of benchmarks to confirm this observation.

The final application presents useful tools that are needed in a practical scenario. Specifically, the tools presented allows the entity with the input to verify that the model used in the computation was previously certified by a trusted party. This in particular would be needed in cases where the fairness of the model could be questioned. In addition, we also present techniques for proactive secret-sharing which is needed if the shared Machine Learning model needs to be moved from one set of server to another.

Resumé

Maskinlæring bliver brugt i et større og større omfang. I mange af de situation hvor maskinlæring avendes, er enheden der leverer input en anden enhed, ejet af en anden person, end den enhed der leverer maskinlæringsmodellen. Mange af disse anvendelser sker desuden på data der er følsomt, såsom personlige billeder eller tekst, og metoder hvorpå vi kan forblive i kontrol over vores data, samtidig med at vi kan gøre brug af maskinlærning, er derfor vigtigt.

Sikker flerparts beregning (på engelsk MPC, fra "Secure Multiparty Computation") giver os en attraktiv løsning til netop dette problem, og denne afhandling præsenterer tre konkrete måder hvorpå MPC kan anvendes til sikker maskinlærning.

I den første anvendelse, demonstreres der protokoller til sikker flerparts beregning med aktiv sikkerhed, og med egenskaber der er specielt attraktive til sikker maskinlærning. Protokollerne vi præsentere er praktisk effektive i specielt det scenario vi betragter: navnlig aktiv sikkerhed med en ærlig majoritet til beregning over ringe. Vi præsentere to protokoller: en der virker for 3 partier der er mindst ligeså effektiv som tidligere research, samt en for et vilkårligt antal partier der er den første af sin slags.

I den anden anvendelse kigger vi på metoder hvorved maskinlæringsmodeller kan anskaffes, således at disse kan evalueres sikkert og ikke mindst effektivt. Sikker maskinlærning (evaluering af en allerede trænet maskinlæringsmodel på et hemmeligt input) kræver ofte justering med hensyn til enkodning af værdier, samt hvilke ikke-lineærer funktioner der anvendes. Vi observerer, at modeller der er designet til evaluering på f.eks. smartphones, også virker specielt godt når vi evaluerer dem sikkert, og vi præsentere en lang række eksperimenter der underbygger denne observation.

I den sidste anvendelse præsentere vi brugbare redskaber der vil være nødvendige i praktiske scenario. Lidt mere præcist, så præsentere vi protokoller der kan bruges til at garanterer, at en model der avendes i en sikker beregning, tidligere er blevet certificeret af en betroet tredjepart. Derudover præsentere vi også protokoller for såkaldt "proactive secret-sharing" der er nødvendige hvis en model skal flyttes fra en gruppe af serverer til en anden.

Acknowledgments

To everyone at the cryptography group at Aarhus; everyone I have interacted with at conferences, internships, workshops, visits, events or travels; everyone I have worked with, in person or remotely; to my advisor; to my family, and to my friends. Thank you all for making these last three or so years so enjoyable.

Anders Dalskov,
Aarhus, November 11, 2020.

Contents

Abstract i

Resumé iii

Acknowledgments v

Contents vii

I Introduction 1

1 Background 3

1.1 Secure Multiparty Computation 4
1.2 Secure Inference 7

2 Techniques for Secure Inference 11

2.1 Fast Actively Secure Computation over rings 12
2.2 Models for Secure Inference 13
2.3 Certifying inputs 14
2.4 Additional Research Activity 15

II Publications 17

3 Passive-to-active Compiler 19

3.1 Introduction 20
3.2 Preliminaries and Definitions 22
3.3 Building Blocks and Sub-Protocols 24
3.4 The Main Protocol for Rings 26
3.5 Instantiation for n parties 30
3.6 Implementation and Evaluation 36

4 Secure Quantized Inference 41

4.1 Introduction 42
4.2 Deep Learning and Quantization 48

4.3 Quantized CNNs in MPC 52
4.4 Implementation and Benchmarking 60
4.5 Conclusions 67

5 LSS Homomorphisms and Applications 69
5.1 Introduction 70
5.2 LSS Homomorphisms and Bilinear Maps 72
5.3 Instantiations 75
5.4 Digital Signatures in MPC 79
5.5 Applications to Proactive Secret Sharing 81
5.6 Applications to Input Certification 86
5.7 Implementation and Benchmarking 87

Appendix 91

Appendix to Chapter 5 93
Proof of Lemma 1 93
Instantiation for 3 parties 96
Appendix to Section 3.5 - Shamir-SS Instantiation 98

Appendix to Chapter 6 103
Related work on secure inference 103
MPC Protocols 104
Extended results 105

Appendix to Chapter 7 109
Bilinear maps for MPC 109
Proofs 110
Shamir Secret-Sharing 112
Optimizations 115
Communication Complexity of CHURP 116
Secure Computation over Elliptic Curves 117

Cats or Croissants? 121

Bibliography 123

Part I

Introduction

Chapter 1

Background

Machine Learning is a general term for algorithms that can tell us something useful without being explicitly programmed to do so. As a silly example, and to explain the question in the title, one use of Machine Learning is to tell us whether a specific picture depicts a cat or a croissant. Because there is an almost infinite amount of pictures of both cats and croissants, it would be infeasible to construct an algorithm that decides its answer by direct inspection (i.e., an algorithm which says “if the input is this picture, then it’s a cat”). A properly trained Machine Learning model, such as a Convolutional Neural Network, provides us with a means to answer such questions with impressive accuracy.[1]

Silly examples aside, Machine Learning has been shown to be useful for solving a wide variety of problems that were previously thought to be best left for a human. Machine Learning has for example been used to write scientific books [143] (in particular an overview of the latest and greatest research in Lithium-ion batteries); playing video games [141] or board games [137]. Machine Learning has also been demonstrated to be capable of identifying various cancers from images such as skin cancer [68] or breast cancer [136]. Machine Learning can even generate meaningful text in multiple different languages, both artificial and natural [33] (see also [92]).

Training and designing Machine Learning models is a long and often strenuous process, not least in part due to the large amount of data that is needed. For example, [68] used a dataset of more than 120 000 images, and [33] used a corpus of almost a trillion words. Even the process of running the inference step (that is, using the model after it has been trained) can be computationally intensive on devices with less computing power. Indeed, modern Convolutional Neural Networks that perform well in e.g., image recognition tasks often require computing in the order of billions of arithmetic instructions. While research into models which work well on more consumer friendly devices (such as smart-

[1] See `https://anderspkd.github.io/blog/cats_or_crossaints.html`, or just Appendix 5.7 for a picture of cats and croissants that look strikingly similar.

phones) have been quite active, another typical way of deploying Machine Learning is as an outsourced service, also called *Machine-Learning-as-a-Service*, or MLaaS for short. In the MLaaS setting, the model is stored at a remote provider who runs the inference tasks on behalf of its users. Most major tech companies such as Microsoft, Google, Amazon and IBM all provide MLaaS today.

The widespread use of Machine Learning, and the fact that the entity with the model (who we will call the *model owner*) is not always the same as the entity with the input (the *input owner*) raises questions with respect to privacy. Consider for example a group of hospitals that wants to use a MLaaS provider in order to help practitioners better detect skin cancer in their patients [68]. During an examination, an image x of a patient's skin is sent to the provider who runs a Machine Learning algorithm on M (that was trained by the provider using their proprietary model with data supplied by the hospital) and x, and returns a prediction (e.g., cancer or no-cancer with confidence so-and-so).

One can imaging that this would help the doctors, since the Machine Learning algorithm effectively provides a second opinion. However, the issue in this scenario is that the user needs to send their sensitive input x to a third party. The prediction algorithm itself is typically lightweight enough that it can be run on a desktop computer, so one can hope that we can solve this privacy issue by just storing the model M at every practice. This does provide privacy for the patient, but what about the intellectual property of the provider? Indeed, training a model is an expensive process and we have just given the result of this task to every doctor that use this system. Moreover, having to replicate M onto a lot of devices is likely to create problems when M eventually have to be updated. These issues create an impasse which, in order to be overcome, seemingly implies that we must either sacrifice the privacy of the model owner or the input owner.

Luckily, this is not so, and Cryptography, specifically "Secure Multiparty Computation" (abbreviated as *MPC*), provides us with a way to overcome this dilemma. There are, however, a lot of different questions that arise when MPC is used for secure inference, such as which protocol to use, which models to use and how to deal with issues not strictly related to privacy. Secure Inference using MPC or other techniques for secure computation (such as homomorphic encryption or gabled circuits) have received a lot of attention from the research community in recent years [42, 109, 116–118, 124, 131, 142] as well as industry [52, 70]. This thesis contributes to this research by providing solutions for several practical problems that arise in the context of secure inference.

1.1 Secure Multiparty Computation

Although MPC has its origin in research from the 80s [22, 43, 82, 145], it was not until the late 2000s before it saw use in real life [28]. Subsequent years have

seen protocols of increasing speed (e.g., [9] demonstrates a protocol for boolean computation with a throughput in excess of 1 billion boolean gates per second) as well as an increase in interest from outside academia. Indeed, there are now multiple companies that develop and provide MPC based solutions [1–3].

MPC is a technique that allows a set of parties to perform a computation with some guarantee with respect to privacy. More precisely: an MPC protocol for a function f is an interactive process[2] between n parties $P_1, \dots, P_n$, with inputs $x_1, \dots, x_n$ such that, in the end, every party learns $z = f(x_1, \dots, x_n)$ and *nothing else.*

To make this idea of "*nothing else*" more precise, we will introduce an adversary $\mathcal{A}$ that is allowed to take control of $t < n$ of the parties. Restrictions on the magnitude of t as well as the actions of the corrupted parties are both important parameters of any MPC protocol, and plays a direct role in both their efficiency as well as what is possible to achieve in terms of security.

Corruption threshold. The *corruption threshold* t of a protocol decides the number of parties that $\mathcal{A}$ is allowed to take control over, and we will draw a clear line between the case where $t < n/2$ and the case where $t < n$. Given an MPC protocol Π that is secure when $\mathcal{A}$ corruptions at most $t < n/2$ of the parties, we say Π is secure with an *honest majority.* Similarly, if Π is secure when $\mathcal{A}$ is allowed to control $t < n$ of the parties, then Π is secure with a *dishonest majority.*

The honest majority setting is a weaker threat model than the dishonest majority one. Indeed, an honest majority implies that not *too many* parties collude (for example, with $n = 3$, an honest majority implies that no one colludes). No such assumption is needed for the dishonest majority case. It is therefore not surprising that protocols in the dishonest majority setting, as they have to be secure against a stronger adversary, are often less efficient than their honest majority counterpart. Indeed, we can compare the results presented in Chapter 4 in Table 4.3 and Table 4.4 and see that the difference between these two settings (all other things equal) is several orders of magnitude in terms of running time.

Corruption type. Another parameter is with respect to the type of corruption that $\mathcal{A}$ makes. In particular, whether or not the corruptions are *passive* or *active.* In the former case, parties under the control of $\mathcal{A}$ are assumed to follow the protocol, while in the latter case, parties can behave arbitrarily. Clearly, security against a passive adversary is easier to obtain as in this case the protocol simply needs to be private (i.e., not leak anything besides what can be learned from the inputs and outputs). The case of active security is

[2]Of course, it is possible to think of a function f that always outputs the same thing, and therefore one for which no interaction is needed. These are likely not very interesting though, and so we will ignore them.

potentially more complex as we must ensure that the corrupted parties do no introduce errors in the computation.

In terms of efficiency, the gap between passive and active security is highly dependent on the corruption threshold as well. Looking again at Table 4.3 and Table 4.4 in Chapter 4 the gap between the passive and active protocols are a lot smaller for protocols with an honest majority, compared to protocols with a dishonest majority. Moreover, the protocol presented in Chapter 3 has an overhead of roughly $2\times$ that of the passive protocol, and other techniques (that are computationally costly) can reduce this even further [32].

Adversary power. The final parameter with which we will parameterize an MPC protocol is with respect the power of $\mathcal{A}$. In particular whether it is computationally bounded or not. If $\mathcal{A}$ is computationally bounded, then we can make use of computational assumptions such as those related to the hardness of Discrete Logarithms in certain groups, or the assumption underlying RSA.

Security

Security of an MPC protocol is shown via simulation, where the idea goes as follows: Suppose we had access to a trusted third party $\mathcal{F}$ which receives the inputs from all the parties, performs the computation and returns the output. This trusted third party, or *ideal functionality*, is clearly secure since all that is ever revealed is the output. However, $\mathcal{F}$ likely does not exist in the real world and so we will need a practical stand in.

the Universal Composability (UC) framework [36] provides a way to formally show that a practical protocol Π can act as a stand in for $\mathcal{F}$. Put differently, instead of using $\mathcal{F}$ (which we can't, as it does not exist) we will show that some practical protocol Π behaves identically to $\mathcal{F}$. If no one can tell $\mathcal{F}$ and Π apart, then Π must be as secure as $\mathcal{F}$.

Let $\mathcal{Z}$ denote the adversary (or, more accurately, the environment in which a protocol is run). Consider as well two additional adversaries: $\mathcal{A}$ and $\mathcal{S}$ where the former is a real world adversary and the latter an ideal world adversary. We say that Π is secure if $\mathcal{Z}$ cannot distinguish between a world in which we run Π with the adversary $\mathcal{A}$, from the world where we run $\mathcal{F}$ with the adversary $\mathcal{S}$. This can be expressed succinctly by writing $\texttt{EXEC}_{\mathcal{Z},\mathcal{F},\mathcal{S}} \approx \texttt{EXEC}_{\mathcal{Z},\Pi,\mathcal{A}}$.

Given a definition of $\mathcal{F}$ and Π we go about showing this indistinguishability by describing how $\mathcal{S}$ is supposed to act. The strategy most often used is for $\mathcal{S}$ to use $\mathcal{A}$ internally in order to mimic attacks in the ideal world that $\mathcal{A}$ would carry out in the real world.

One of the attractive features of the UC framework is that security composes. If we have shown that Π behaves as $\mathcal{F}$, and that we now want to say something about another protocol Π' which uses Π as a subprotocol. As before, we would need to show that Π' is indistinguishable from some functionality $\mathcal{F}'$. However, we do not need to consider some combination of Π and Π', and we can instead

consider a hybrid world in which Π' is allowed to invoke $\mathcal{F}$. This means that security can be proven in a piecemeal fashion, which, given the complexity of many MPC protocols, is a very attractive feature.

Secret-Sharing

MPC protocols typically rely on secret-sharing, of which two schemes in particular are common: additive secret-sharing and Shamir secret-sharing [135].

Additive secret-sharing proceeds as follows: Let K be a field and $s \in K$ the secret. In order to share s among n parties, we pick $s_1, \ldots, s_n$ uniformly at random from K such that $s = \sum_{i=1}^{n} s_i$ and give s_i to party P_i. We use the notation $[\![s]\!]$ to mean that each party holds s_i. Observe that this sharing is linear. More precisely, given $[\![x]\!]$ and $[\![y]\!]$, parties can locally compute $[\![x + y]\!]$ by simply adding their respective shares locally. Multiplication is more tricky, and for additive secret-sharing techniques such as described by Beaver [20] can be used (so called multiplication triples).

Additive secret-sharing has a threshold of $n - 1$; that is, any subset of size $n - 1$ reveals no information about s. To get more control over this threshold (for example if we wish that any majority of the parties can recover s) we can use Shamir secret-sharing. The idea is as follows: To share $s \in K$, we pick a degree $t - 1$ polynomial $f \in K[X]$ at random such that $f(0) = s$, and give to party P_i the evaluation at i, that is $s = f(i)$. Notice that this scheme too is linear, since parties can locally add their shares to obtain a sharing of the sum. As before, multiplication presents a bit of an issue, but can be overcome for example by using the same trick as above (triples) or by using a random double sharing [58].

1.2 Secure Inference

Recall the two parties involved in secure inference: one is the model owner who has a model M, and the other is the input owner who has an input x. The input owner wishes to learn something about x by way of Machine Learning inference using the model owner's model M. However, and as should be clear by now, none of them wish to reveal their input to the other party.

This is, in a nutshell, what is meant by secure inference, and MPC seems like an obvious candidate for tackling this problem: The model owner and input owner run an MPC protocol that evaluates the inference algorithm, and which at the end outputs the result of the inference to the model owner. The security of the MPC protocol guarantees that the input owner only ever sees the result of the inference, and that the model owner doesn't see anything. (Here it is important to remark that MPC does not protect against e.g., model stealing attacks [140]. Indeed, these attacks abuse precisely that the input owner sees the result of the inference.)

As mentioned in the previous section, two party MPC (which implies a dishonest majority), is an expensive process. And it might be prohibitively expensive for computations such as inference, which often requires 100s of millions of multiplications. Indeed, all the most efficient secure inference protocols so far rely on an honest majority [54, 109, 124, 142].

There is two ways to go about the honest majority setting. The first is to involve an additional party that both the model owner and input owner agree wont collaborate with the other.

The other approach involves so called *outsourced computation*, and is perhaps the setting that best fits the MLaaS scenario for secure inference. Outsourced computation, as the name implies, is a process whereby the parties with the inputs (the model owner and input owner) outsource the actual computation to a collection of third parties which perform the actual computation. More precisely, the model owner and input owner both share their inputs towards a collection of computing parties $P_1, \ldots, P_n$.[3] The parties $P_1, \ldots, P_n$ then run the secure computation on behalf of the model owner and input owner, and return the result to the latter. This sort of model has been used in practice before: e.g., for the danish sugarbeet auction [28], or when computing wage statistics for the Boston area [46].

There are several attractive features of the outsourced computation model. First, it permits participation of users even if these do not have a powerful computer, and second the users do not have to be online during the computation: they only have to show up at the beginning to provide input, and at the end to obtain the result. Moreover, because the computing parties can be a fixed set of computers, it is easier to imagine that these are very well connected, and so the communication overhead induced by the MPC is less of an issue.[4] Finally, certain protocols enjoy an enhanced notion of security in the outsourced computation setting. Most notably the protocol of Araki et al. [8] which is a passively secure protocol, *but* which in addition preserves privacy against a malicious adversary in the outsourced computation setting (which the authors call client/server model). This protocol underlines e.g., SecureNN [142] which is a highly efficient protocol for secure inference.

Secure inference isn't just running the inference algorithm securely, however, and several issues arise when inference is run in MPC.

Floating point numbers. A machine learning model M can be viewed as a large collection of real valued parameters which in turn implies that the inference is performed using floating point numbers. This is of course not an issue in the clear, but becomes problematic in MPC that only operate on

[3] Notice that there is no requirement that this sharing happen at the same time. E.g., the model owner could share its model once, and then multiple different users could also use the system at different later times.

[4] Indeed, the high efficiency of MPC typically (but not always) assume that parties are connected in a way where throughput and latency are minor concerns.

integers (in either a field $\mathbb{Z}_p$ for a prime p, or ring $\mathbb{Z}_{2^k}$). The way real values are handled in MPC is to treat them as fixed point values. More precisely, for a value $\alpha \in \mathbb{R}$, the secure computation is performed on $a \in \mathbb{Z}$ where this value is such that $\alpha \approx a2^{-\ell}$ for a fixed scale ℓ.

Addition of two fixed point values is straight forward since, for $\alpha, \beta \in \mathbb{R}$ encoded as a, b, we have that $\alpha + \beta \approx a2^{-\ell} + b2^{-\ell} = (a + b)2^{-\ell}$.

Multiplication is a bit more tricky. In particular, $\alpha\beta \approx (ab2^{-\ell})2^{-\ell}$ and so the fixed point result is obtained by multiplying a and b as integers and truncating the result by ℓ bits. There has been a number of protocols for truncation in recent years; e.g., [34, 54, 107, 117, 118, 124, 127, 142] all contain protocols for truncation. The most efficient approach is to do what is called a probabilistic truncation, in which the result might be wrong by a small amount.

Non-linear functions. Another issue we face when performing secure inference is the non-linear functions. While Machine Learning is mostly linear algebra[5] in the form of matrix multiplications or convolutions (which can be expressed as matrix multiplications), it also involves a significant amount of functions that are not linear. In more detail, evaluating a Neural Network is essentially a sequence of non-linear functions f_1 to f_k where the i'th function receives $z_{i-1} \cdot M_i$ as the input, where z_{i-i} is the output of the previous function (or layer) and M_i is the parameters from the model associated with the i'th function.

Convolutional Neural Networks typically employ functions such as rectified linear units (defined as $\mathsf{ReLU}(x) = max(0, x)$), sigmoid, hyperbolic tanget or Sign.

Prior work deals with these functions in a variety of ways. For ReLU, efficient bit extraction is required (in order to compute max, which requires a comparison). ABY3 [117] and BLAZE [124] both achieve this by computing a parallel prefix adders. SecureNN [142] on the other hand use a property of the rings $\mathbb{Z}_{2^k}$ and $\mathbb{Z}_{2^k-1}$ in order to obtain a protocol which computes the most significant bit of a value, by way of finding the least significant bit of a slightly different value.

Other activation functions are often approximated using polynomials, although previous work such as SecureML [118] have shown the polynomial used needs to be of a fairly high degree before a good accuracy can be obtained.

Other works such as Delphi [116] explores a mix of simpler, more efficient custom activation functions (such as the function $x \mapsto x^2$) together with standard activation functions (such as the ReLU activation function).

[5] `https://xkcd.com/1838/`

Chapter 2

Techniques for Secure Inference

The following chapter outlines and introduces the research I have been engaged in during my roughly 3 years of study. As the chapter title implies, the focus will be on techniques that are useful for (practical) secure inference. However, this area of research is by no means the only area I have been involved in, and I have been involved in projects on personal cloud storage [53, 57],[1] machine learning [54], honest majority MPC [6, 55], and threshold public-crypto schemes [11, 56].

That being said, of these works, this thesis focuses on three in particular:

- *An Efficient Passive-to-Active Compiler for rings* [6]. This work presents a highly efficient compiler that turns a passively secure protocol into an actively secure one with very little overhead. This compiler works for rings, and preserve some attractive features of the passive protocol in the context of Machine Learning.

- *Secure Inference of Quantized Neural Networks* [54]. This work examines certain types of Convolution Neural Network models which (1) can be trained with Tensorflow, and (2) be evaluated *without tweaking* in MPC. We moreover provide optimized protocols for truncation, as well as a very large set of benchmarks that give a unique insight into the efficiency of secure inference with various MPC protocols.

- *LSS Homomorphisms and Applications to Secure Signatures, Proactive Secret Sharing and Input Certification* [11]. This final work formalizes techniques for running any pairing based scheme with any MPC protocol. We apply our technique to two concrete applications: proactive secret-sharing and input certification.

[1] Note that [57] was done during my Masters; and [53] builds on some of the observations made therein.

The next couple of sections goes a bit more into the details of these three works, and the chapter concludes with a short overview of the other research I have done.

2.1 Fast Actively Secure Computation over rings

Protocols for secure computation over rings, in particular $\mathbb{Z}_{2^k}$, have received a good amount of attention in recent years—and for good reason. Operations in the ring $\mathbb{Z}_{2^{32}}$ or $\mathbb{Z}_{2^{64}}$ correspond precisely to the operations on fixed width integers that is supported natively on a modern computer and so are very fast. This is in contrast to the more usual setting for MPC where the algebraic structure used is $\mathbb{Z}_p$ with p being a prime. Notably, even if p is chosen as a prime which allow a fast reduction procedure (such as a Mersenne prime) this reduction nevertheless incurs a small overhead that is avoided by working over a ring of similar size.

The protocol compiler we present in Chapter 3 provides a technique that turns a passively secure protocol into an actively secure one, where the overhead is very small in concrete terms. In a nutshell, the overhead is only roughly twice that of the passive protocol.

The idea behind our compiler is similar to the one in [45] where two instances of the passive protocol are run alongside each other, but where one instance is randomized by an unknown random value. That is, at every point during the computation, each party holds shares of $[\![x]\!]$ and $[\![r \cdot x]\!]$ where x is the actual wire value and r a random secret-shared value. The adaption is not straightforward, however, and there are a number of subtle issues that arise when working over a ring as opposed to over a field.

The efficiency of our compiler is demonstrated by providing two concrete instantiations: One based on replicated secret-sharing ala [8] for 3 parties; the other is based on Shamir secret-sharing over rings based on the protocol presented in [5]. For the 3 party instantiation, we perform an experimental comparison against a collection of recent very efficient protocols also for 3 parties as presented in [67] and conclude that ours is as fast, and even faster in some scenarios. For the n instantiation, we remark that this paper was the first to report any kind of implementation for general MPC over rings with an honest majority and as such there is no direct work to compare against. We therefore compare it against the field based protocol in [45] and show that it is comparable in terms of efficiency. It is worth remarking that the protocol of [5] that we base ours on, induce a $\log n$ overhead, since computation have to be performed over an extension of $\mathbb{Z}_{2^k}$ in order for interpolation to work.

Finally, both our protocols enjoy the particularly attractive property of having free inner products. That is, computing $\langle \mathbf{x}, \mathbf{y} \rangle$ securely has the cost of only a single secure multiplication regardless of the length of $\mathbf{x}$ and $\mathbf{y}$. For Machine Learning, where the bulk of the computation requires operations

on matrices, this is indeed a nice feature to have, and has been pointed out in several subsequent works such as e.g., BLAZE [124]. Furthermore, our 3 party instantiation is the first protocol that obtains this property for active security without relying on preprocessing (e.g., [124] requires function dependent preprocessing in order to obtain cheap inner products).

Bibliography information

Mark Abspoel, Anders Dalskov, Daniel Escudero, and Ariel Nof. An efficient passive-to-active compiler for honest-majority mpc over rings. Cryptology ePrint Archive, Report 2019/1298, 2019. `https://eprint.iacr.org/2019/1298`. Currently under submission.

2.2 Models for Secure Inference

Prior works on secure inference often only treats the question of how to obtain the models that will be evaluated in a superficial manner. Since secure inference ultimately requires some modifications—like using a fixed point encoding, or altering the activation functions—how to easily obtain a model that can evaluated securely is nevertheless an important question to ask.

In Chapter 4 we look towards so-called "quantized" models as candidates for models which can evaluated efficiently by an MPC protocol *without* having to make any tweaks to the model. Quantization is a technique from Machine Learning which attempts to design models which are lightweight, in particular so they are efficient to evaluate on devices that a resource-wise constrained. Quantization is moreover implemented and supported in most major Machine Learning frameworks such as Tensorflow or Pytorch.[2] As it turns out, many of the optimizations that are made to make inference efficient on e.g., smartphones, also help making inference efficient in MPC.

We present a generic way of evaluating these quantized models, that is, a way that does not depend on the underlying MPC protocol. We then benchmark inference of a class of large models for image prediction that are being used in practice (so-called MobileNets models that were developed by Google and which are used in many mobile applications). Each of our benchmarks are run with essentially all possible combinations of parameters for the MPC: that is, honest and dishonest majority, passive and active security, as well as over a field and over a ring. These experiments show first and foremost that quantized models are useful and efficient for secure inference, but also show more generally an interesting insight into the trade-offs between different MPC threatmodels.

Finally, we also provide optimized protocols for the most efficient setting (honest majority with passive security for computation over a ring) and bench-

[2]While Chapter 4 focuses exclusively on models than can be obtained from Tensorflow, other popular frameworks such as pytorch and MXNet also support the same kind of quantization and so our techniques apply to models from these frameworks as well.

mark this particular instance against the previous fastest protocols for secure inference from CrypTFlow [109], and we show that we outperform it by a significant margin.

Bibliography information

Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *Proceedings on Privacy Enhancing Technologies*, 2020 (4): 355 – 375, 01 Oct. 2020. doi: https://doi.org/10.2478/popets-2020-0077 URL `https://content.sciendo.com/view/journals/popets/2020/4/article-p355.xml`.

2.3 Certifying inputs

Finally, Chapter 5 presents techniques that are useful for practical deployments of secure inference (and other problems as well).

This work presents techniques for computing any bilinear function in MPC—in particular, this technique allows one to securely compute schemes that rely on cryptographic pairings. Chapter 5 shows how to perform such computations in a general way, i.e., independent of the underlying secret-sharing scheme (so long as the secret-sharing scheme is linear and defined over some vector space). We show first how these techniques can be used to compute Pointcheval-Sanders signatures [126] and then present two applications, one of which is of particular interest to the topic of this thesis. The first application is an dynamic proactive secret-sharing scheme with abort which outperforms the current comparable state of the art [113]; the second application is input certification for MPC which, for a large number of messages, outperform the previous only scheme which performs input certification with signatures [26].

Fairness is currently an active research area (see e.g., [17]), and fairness of Machine Learning models does not stop being a concern simply because the model is now hidden. In fact, MPC makes it quite explicit that the model owner might try to cheat and so fairness of the model they input should be ensured. The input certification application we demonstrate in this work provides a solution for this problem. The experiments we report on can be used to get an idea of how costly certifying a Machine Learning model would be. For example, the models we consider in [54] have between 0.5 and 4.2 million parameters. Thus the implementation we report on shows that we can certify the smallest model in around 80 seconds, and the largest in around 10 minutes. It is worth noting that this certification only has to happen once: after the computing parties have obtained (and verified) a secret-shared model, they can use it for many subsequent computations.

Finally, the proactive secret-sharing application we show is also useful for secure inference. Indeed, if one or more of the computing parties need to be

replaced, then our proactive secret-sharing protocol can be used to transfer the secret-shared model without involving the model owner.

Bibliography information

Diego Aranha, Anders Dalskov, Daniel Escudero, and Claudio Orlandi. Lss homomorphisms and applications to secure signatures, proactive secret sharing and input certification. Cryptology ePrint Archive, Report 2020/691, 2020. `https://eprint.iacr.org/2020/691`. Currently under submission

2.4 Additional Research Activity

The above works are not the only research I have been involved in, and the final part of this chapter provides short summaries of the some of my other research.

Threshold ECDSA and DNSSEC

The techniques developed in [11] were based on tricks that was used in another work of mine [56]. In [56] we use the idea of performing MPC "in the exponent" to obtain highly efficient yet generic protocols for threshold ECDSA, and demonstrate an application to DNSSEC. (The term generic here is the same as in [54], i.e., our threshold ECDSA scheme does not rely on particular properties of a specific MPC protocol.) We further make the interesting observation in [56] that much of the material needed when computing a threshold ECDSA signature can be preprocessed, which results in very fast signing times when amortization is taken into account.

Bibliography information

Anders Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. Securing DNSSEC Keys via Threshold ECDSA from Generic MPC. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve Schneider, editors, *Computer Security – ESORICS 2020*, pages 654–673, Cham, 2020. Springer International Publishing. ISBN 978-3-030-59013-0.

Circuit Amortization Friendly Encodings

In [55] we present a number of encoding schemes in order to make computation over $\mathbb{Z}_{2^k}$ with Shamir secret-sharing more efficient. Shamir secret-sharing over $\mathbb{Z}_{2^k}$ requires working over an extension of degree $\log n$ in order for interpolation to work. The encodings we present here makes use of "the extra space" in this extension in order to e.g., compute several circuits in parallel, or compute small inner products. We moreover present an efficient statistically secure check

of random double-shares. We implement all of our encodings and show that they outperform the double-share generation protocol of [5], which was also implemented.

Bibliography information

Anders Dalskov, Eysa Lee, and Eduardo Soria-Vazquez. Circuit Amortization Friendly Encodings and their Application to Statistically Secure Multiparty Computation. Cryptology ePrint Archive, Report 2020/1053, 2020. `https://eprint.iacr.org/2020/1053`. Will appear at ASIACRYPT 2020.

Two-Factor Secure Personal Cloud Storage

In [53] we develop a simple solution for personal cloud storage with two-factor security. Our system permits the user to lose control over one of their devices, without denying them access. That is, we obtain a system that provides both a notion of two-factor security as well as recovery. The idea, in a nutshell, is to involve the storage provider in a way which effectively gives us a system of three parties where no one colludes.

Bibliography information

Anders Dalskov, Daniele Lain, Enis Ulqinaku, Kari Kostiainen, and Srdjan Capcun. 2FE: Two-factor Encryption for Cloud Storage. Currently under submission.

Part II

Publications

Chapter 3

An Efficient Passive-to-Active Compiler for rings

Mark Abspoel[†], Anders Dalskov[‡], Daniel Escudero[‡], Ariel Nof[*]

† CWI, Amsterdam
‡ Aarhus University, Denmark
∗ Bar-Ilan University, Israel

Abstract. Multiparty computation (MPC) over rings such as $\mathbb{Z}_{2^{32}}$ or $\mathbb{Z}_{2^{64}}$ has received a great deal of attention recently due to its ease of implementation and attractive performance. Several actively secure protocols over these rings have been implemented, for both the dishonest majority setting and the setting of three parties with one corruption. However, in the honest majority setting, no *concretely* efficient protocol for arithmetic computation over rings has yet been proposed that allows for an *arbitrary* number of parties.

We present a novel compiler for MPC over the ring $\mathbb{Z}_{2^k}$ in the honest majority setting turns a semi-honest protocol into an actively secure protocol with very little overhead. The communication cost per multiplication is only twice that of the semi-honest protocol, making the resultant actively secure protocol almost as fast.

To demonstrate the efficiency of our compiler, we implement both an optimized 3-party variant (based on replicated secret-sharing), as well as a protocol for n parties (based on a recent protocol from TCC 2019). For the 3-party variant, we obtain a protocol which outperforms the previous state of the art that we can experimentally compare against. Our n-party variant is the first implementation for this particular setting, and we show that it performs comparably to the current state of the art over fields.

3.1 Introduction

Multiparty computation (MPC) is a cryptographic tool that allows multiple parties to compute a given function on private inputs whilst revealing only its output; in particular, parties' inputs and the intermediate values of the computation remain hidden. MPC has by now been studied for several decades, and different protocols have been developed throughout the years.

Most MPC protocols are "general purpose", meaning that they can in principle compute *any* computable function. This generality is typically obtained by representing the function as an arithmetic circuit modulo some integer p. Note that implied in this representation, is a set of integers on which computation can be performed. Traditionally, MPC protocols are classified as being either *boolean* or *arithmetic*, where the former have $p = 2$ and the latter has $p > 2$. However, most of the existing arithmetic MPC protocols, independently of their security, require the modulus to be a prime (and for some protocols this prime must be large) [21, 23, 45, 59, 75, 104, 111].

It was only recently that practical protocols in the arithmetic setting for a non-prime modulus were developed. The SPD$\mathbb{Z}_{2^k}$ protocol securely evaluates functions in the dishonest majority case [51], while several other works focus on honest majority case for small number of parties [5, 45, 67, 75]. Computation over $\mathbb{Z}_{2^k}$ is appealing due benefits in performance over computation over fields, as verified in [62]. These benefits are partly due to the fact that arithmetic over rings like $\mathbb{Z}_{2^{32}}$ or $\mathbb{Z}_{2^{64}}$ can be implemented more efficiently in modern hardware than arithmetic over $\mathbb{F}_p$, which requires a software implementation for reduction modulo p. Also, non-arithmetic operations like comparison and truncation become simpler and more efficient in this setting [62, 117]. Though results are recent, MPC over rings has already been used in applications like privacy-preserving machine learning and secure evaluation of neural networks [54, 117, 118, 142]. However, to this date, no concretely-efficient protocol that works for any number of parties has been proposed in the honest majority setting.

Our Contributions

In this work, we develop highly efficient protocols over $\mathbb{Z}_{2^k}$ by presenting a generic compiler that transforms a passively secure protocol for computation over $\mathbb{Z}_{2^{k+s}}$ in the honest majority setting, to a protocol over the ring $\mathbb{Z}_{2^k}$ that is actively secure with abort and provides roughly s bits of statistical security. Summarizing our contributions:

- Our compiler requires the passively secure protocol to be secure up to an additive attack. That is, an active adversary can at most introduce an additive error in the passively secure protocol. This was shown to be the case for multiple well-known protocols over fields in [78], a result which we extend in our paper to recent protocols over rings.

- Our compiler is highly efficient and the overhead is essentially just twice that of the passively secure protocol. More precisely, each multiplication just needs to be evaluated twice.

- Our compiler preserves all the properties of the passive secure protocol. In particular we obtain the first actively secure protocols where the cost of dot products is *independent* on their length without relying on an expensive function dependent preprocessing such as is the case for prior work [41, 67, 75, 124].

- Finally, we provide two instantiations and show through experiments that they are concretely efficient: One for 3 parties based on replicated secret sharing, and one for n parties based on the recent by Abspoel et al. [5].

  - Our 3 party instantiation is shown experimentally as well as theoretically to outperform current state of the art that deal with 3 party computation [67, 124].
  - Our n party instantiation is the first *practical* (as in shown experimentally to be concretely efficient) example of such a protocol for $\mathbb{Z}_{2^k}$ with active security and an honest majority. The protocol from [5] requires $3(k + s) \log n$ bits per multiplication in the online phase; however we describe a novel optimization that removes the $\log n$ factor that might be of independent interest. We compare our protocol against [45], which is a recent and highly efficient protocol for fields and show here that our protocol is comparable in terms of efficiency.

Outline. Section 3.2 introduces some of the definitions we will be needing and Section 3.3 introduces the building blocks we need in our compiler. In Section 3.4 our main protocol (i.e., our compiler) is presented, as well as the formal statements of security, the proofs of which are placed in the Appendix due to space constraints (Appendix 5.7). We then present the n party instantiation in Section 3.5. Our three party instantiation is located in Appendix 5.7; as it is technically less interesting than our n party protocol, we chose to put it in the appendix due to the limited space. Finally, Section 3.6 discusses comparison with prior works, as well as presents our experimental results.

Related Work

The only previous general compiler with concrete efficiency over rings, to the best of our knowledge, is the compiler of [60], which was improved by [67]. However, their compiler does not preserve the adversary threshold when moving from passive to active security. In addition, in [60] and [67] the compiler was instantiated for the 3-party case only.

The only concretely efficient protocol for arithmetic computation over rings that works for *any* number of parties is the SPD$\mathbb{Z}_{2^k}$ protocol [51] which was proven to be practical in [62]. This protocol is for the dishonest majority and thus requires the use of much heavier machinery, which makes it orders of magnitudes slower than ours. However, they deal with a more complicated setting and provide stronger security.

In the three-party setting with one corruption, there are several works which provide high efficiency for arithmetic computations over rings. The Sharemind protocol [27] is being used to solve real-world problems but provides only passive security. The actively secure protocol of [75], which was optimized and implemented in [9], is based on the "cut–and–choose" approach and will be favorable when working over small rings. The actively secure three-party protocol of [67] is the closest to our protocol in the sense that they also focus on efficiency for large rings. The overall communication per multiplication gate of their protocol is $3(k+s)$ bits sent by each party, which is higher than ours by $(k+s)$ bits. We provide a detailed empirical comparison with [67] in Section 3.6. Finally, a new promising direction was presented by [32], but their verification step takes several seconds for a 1-million gate over fields, and this is expected to be orders of magnitude worse for rings due to the need of large-degree Galois ring extensions. The protocols of [41, 124] have a slightly overall higher bandwidth than [9], but focus on minimizing online (input-dependent) cost and they tailor their protocols to specific applications for machine learning. Also, [124] uses the techniques from [32] for the preprocessing, so it is unlikely to provide any efficiency in practice.

Finally, it is important to mention that the techniques from [32], which work for 3 parties, can be generalized to multiple parties as a passive-to-active compiler. This has been done in [88] over fields, and it is not hard to see that these techniques can be made to work over $\mathbb{Z}_{2^k}$ by considering large-degree Galois ring extensions, as done in [32]. However, this method is not practical as even small degree extension can be quite expensive, as shown in this work. Furthermore, the round complexity of the passively secure protocol is not preserved by this transformation.

3.2 Preliminaries and Definitions

Notation. Let $P_1, \ldots, P_n$ denote the n parties participating in the computation, and let t denote the number of corrupted parties. In this work, we assume an honest majority, and thus $t < \frac{n}{2}$. Throughout the paper, we use H to denote the subset of honest parties and C to denote the subset of corrupted parties. We use $[n]$ to denote the set $\{1, \ldots, n\}$. $\mathbb{Z}_M$ denotes the ring of integers modulo M, and the congruence $x \equiv y \bmod 2^\ell$ is denoted by $x \equiv_\ell y$.

We use the standard definition of security based on the ideal/real model paradigm [35, 81], with security formalized for non-unanimous abort. This

means that the adversary first receives the output, and then determines for each honest party whether they will receive `abort` or receive their correct output. It is easy to modify our protocols so that the honest parties unanimously abort by running a single (weak) Byzantine agreement at the end of the execution [84]. For simplicity, we omit this step from the description of our protocols. Our protocol is cast in the synchronous model of communication, in which it is assumed that the parties share a common clock and protocols can be executed in rounds.

Linear Secret Sharing and its Properties

Let ℓ be a positive integer. A perfect (t, n)-secret-sharing scheme (SSS) over $\mathbb{Z}_{2^\ell}$ distributes an input $x \in \mathbb{Z}_{2^\ell}$ among the n parties $P_1, \ldots, P_n$, giving *shares* to each one of them in such a way that any subset of at least $t+1$ parties can reconstruct x from their shares, but any subset of at most t parties cannot learn anything about x from their shares. We denote by $\mathsf{share}(x)$ the sharing interactive procedure and by $\mathsf{open}([\![x]\!])$ the procedure to open a sharing and reveal the secret. The share procedure may take also in addition to x, a set of shares $\{x_i\}_{i\in J}$ for $J \subset [n]$ and $|J| \leq t$, such that $\mathsf{share}(x, \{x_i\}_{i\in J})$ satisfies $[\![x]\!] = (x'_1, \ldots, x'_n)$, with $x'_i = x_i$ for $i \in J$. The open procedure may take an index i as an additional input. In this case, the secret is revealed to P_i only. In case the sharing $[\![x]\!]$ is not correct as defined below, $\mathsf{open}([\![x]\!])$ will output $\perp$. An SSS is linear if it allows the parties to obtain shares of linear combinations of secret-shared values without interaction.

Our compiler applies to any linear SSS over $\mathbb{Z}_{2^k}$ that has a multiplication protocol that is secure against additive attacks, as defined in Section 3.2. The only extra, non-standard properties required by our compiler are the following (for a formalization of the requirements of the SSS, see the full version of this work):

Modular Reduction. We assume that the open procedure is compatible with modular reduction, meaning that for any $0 \leq \ell' \leq \ell$ and any $x \in \mathbb{Z}_{2^\ell}$, reducing each share in $[\![x]\!]_\ell$ modulo $2^{\ell'}$ yields shares $[\![x \bmod 2^{\ell'}]\!]_{\ell'}$. We denote this by $[\![x]\!]_\ell \to [\![x]\!]_{\ell'}$.

Multiplication by 1/2. Given a shared value $[\![x]\!]_\ell$, we assume if all the shares are even then shifting these shares to the right yields shares $[\![x']\!]_{\ell-1}$, where $x' = x/2$.[1]

Throughout the entire paper, we set the threshold for the secret-sharing scheme to be $\lfloor \frac{n-1}{2} \rfloor$, and we denote by t the number of corrupted parties. Now we define what it means for the parties to have *correct* shares of some value.

[1]If all the shares $[\![x]\!]_\ell$ are even then these shares may be written as $[\![x]\!]_\ell = 2 \cdot [\![y]\!]_\ell$, which, by the homomorphism property, are shares of $2 \cdot y$. Since these are shares of x as well, this shows that $x \equiv_\ell 2 \cdot y$, so x is even.

Let J be a subset of honest parties of size $t+1$, and denote by $\mathsf{val}(v)_J$ the value obtained by these parties after running the open protocol, where no corrupted parties or additional honest parties participate, i.e. $\mathsf{open}([\![v]\!]^J)$. Note that $\mathsf{val}(v)_J$ may equal $\perp$ and in this case we say that the shares held by the honest parties are not valid. Informally, a secret sharing is correct if every subset of $t+1$ honest parties reconstruct the same value (which is not $\perp$).

Secure Multiplication up to Additive Attacks [78, 79]

Our construction works by running a multiplication protocol (for multiplying two values that are shared among the parties) that is *not* fully secure in the presence of a malicious adversary and then running a verification step that enables the honest parties to detect cheating. In order to achieve this, we start with a multiplication protocols with the property that the adversary's ability to cheat is limited to carrying out a so-called "additive attack" on the output. Formally, we say that a multiplication protocol is secure up to an additive attack if it realizes the functionality $\mathcal{F}_{\mathsf{Mult}}$, which receives input sharings $[\![x]\!]$ and $[\![y]\!]$ from the honest parties and an additive value d from the adversary, and outputs a sharing of $x \cdot y + d$. Since the corrupted parties can determine their own shares in the protocol, the functionality allows the adversary to provide the shares of the corrupted parties, but this reveals nothing about the shared value.

The requirements defined by this functionality can be met by some semi-honest multiplication protocols over $\mathbb{Z}_{2^\ell}$, namely replicated secret sharing and the more recent protocol of Cramer et al. [5], which is an extension of Shamir Secret Sharing to the setting of $\mathbb{Z}_{2^\ell}$. This will allow us to implement this functionality in a very efficient way.

In addition to the above, we consider a similar functionality $\mathcal{F}_{\mathsf{DotProduct}}$ that, instead of computing one single multiplication, allows the parties to securely compute the dot product of two vectors of shares, where the adversary is allowed to inject an additive error to the final output. As in [45], we will show that the functionality can be realized at almost the same cost as $\mathcal{F}_{\mathsf{Mult}}$.

3.3 Building Blocks and Sub-Protocols

Our compiler requires a series of building blocks in order to operate. These include generation of random shares and public coin-tossing, as well as broadcast. Furthermore, a core step of our compiler is checking that a shared value is zero, leaking nothing more than this binary information. This is not easy to instantiate over $\mathbb{Z}_{2^k}$, and we discuss this in Section 3.3. We stress that our presentation here is very general and it assumes nothing about the underlying secret sharing scheme beyond the properties stated in Section 3.2.

$\mathcal{F}_{\mathsf{Rand}}$ – **Generating Random Coins.** We define the ideal functionality $\mathcal{F}_{\mathsf{Rand}}$ to generate a sharing of a random value unknown to the parties. The functionality lets the adversary choose the corrupted parties' shares, which together with the random secret chosen by the functionality, are used to compute the shares of the honest parties. The way to compute this functionality depends on the specific secret sharing scheme that is being used, and we discuss concrete instantiations later on.

$\mathcal{F}_{\mathsf{Coin}}$ – **Generating Random Coins.** $\mathcal{F}_{\mathsf{Coin}}(\ell)$ is an ideal functionality that chooses a random element from $\mathbb{Z}_{2^\ell}$ and hands it to all parties.

$\mathcal{F}_{\mathsf{BC}}$ – **Broadcast with Abort.** With this functionality, a given party sends a message to all other parties, with the guarantee that all the honest parties agree on the same value. Furthermore, if the sender is honest, the agreed value is precisely the one that the sender sent. The protocol may produce abort, and can be instantiated using the well-known echo-broadcast protocol, where the parties echo the message they received and send it the other parties.

$\mathcal{F}_{\mathsf{Input}}$ – **Secure Sharing of Inputs.** This is a functionality that allows a party to distribute consistent shares of its input. This can be instantiated generically by sampling $[\![r]\!]$ using $\mathcal{F}_{\mathsf{Rand}}$, reconstructing this value to the party who will provide input x, and letting this party broadcast the difference $x - r$. The parties can then compute the shares $[\![x]\!] = (x - r) + [\![r]\!]$.

$\mathcal{F}_{\mathsf{CheckZero}}$ – Checking Equality to 0

For our compiler we require a functionality $\mathcal{F}_{\mathsf{CheckZero}}(\ell)$, which receives $[\![v]\!]_\ell^H$ from the honest parties, uses them to compute v and sends `accept` to all parties if $v \equiv_\ell 0$. Else, if $v \not\equiv_\ell 0$, the functionality sends `reject`.

A simple way to approach this problem when working over a field is sampling a random multiplicative mask $[\![r]\!]$, multiply $[\![r \cdot v]\!] = [\![r]\!] \cdot [\![v]\!]$, open $r \cdot v$ and check that it is equal to zero. Clearly, since r is random then $r \cdot v$ looks also random if $v \neq 0$. However, this technique does not work over the ring $\mathbb{Z}_{2^\ell}$: for example, if v is a non-zero even number then $r \cdot v$ is always even, which reveals too much about v. In this section we present a generic protocol to solve the problem of checking equality of zero over the ring, which is unfortunately more expensive and complicated than the protocol over fields described above. On the upside, this check is only called *once* in a full execution of the main protocol and so the complexity of this technique is amortized away. Furthermore, for 3 parties for example, one can get a much more efficient solution, as we show in Section 5.7 in the appendix.

Our general protocol to compute $\mathcal{F}_{\mathsf{CheckZero}}$ is described in Protocol 1. We consider two functionalities, $\mathcal{F}_{\mathsf{CorrectMult}}$ and $\mathcal{F}_{\mathsf{RandBit}}$, that compute correct

multiplications and sample shared random bits, respectively. We discuss in Appendix **??** how to instantiate these functionalities for any secret sharing scheme.

Protocol 1 Checking Equality to 0

Inputs: The parties hold a sharing $[\![v]\!]_\ell$.

1. The parties call $\mathcal{F}_{\mathsf{RandBit}}$ to get ℓ random shared bits $[\![r_0]\!]_\ell, \ldots, [\![r_{\ell-1}]\!]_\ell$.
2. The parties bit-decompose v:
   a) The parties compute $[\![r]\!]_\ell = \sum_{i=0}^{\ell-1} 2^i \cdot [\![r_i]\!]_\ell$.
   b) The parties call $c = \mathsf{open}([\![v]\!]_\ell + [\![r]\!]_\ell)$ and bit-decompose this value as $(c_0, \ldots, c_{\ell-1})$.
   c) The parties locally convert $[\![r_i]\!]_\ell \to [\![r_i]\!]_1$ for $i = 1, \ldots, \ell - 1$.
3. The parties check that all the bits of $v \bmod 2^\ell$ are zero:
   a) The parties use $\mathcal{F}_{\mathsf{CorrectMult}}(1)$ to compute $\bigvee_{i=0}^{\ell-1}([\![r_i]\!]_1 \oplus c_i)$ and open this result.
   b) If the opened value above is equal to 0 then the parties output `accept`. Otherwise they output `reject`.

We have the following proposition.

Proposition 1. *Protocol 1 securely computes $\mathcal{F}_{\mathsf{CheckZero}}$ with abort in the $(\mathcal{F}_{\mathsf{RandBit}}, \mathcal{F}_{\mathsf{CorrectMult}})$-hybrid model in the presence of malicious adversaries who control $t < n/2$ parties.*

3.4 The Main Protocol for Rings

In this section, we present our construction to compute arithmetic circuits over the ring $\mathbb{Z}_{2^k}$. A formal description appears in Protocol 2. Our protocol follows the paradigm of [45], which roughly works by running a "redundant" copy of the circuit where each shared wire value $[\![w]\!]$ is accompanied by $[\![r \cdot w]\!]$ for some global uniformly random r. In [45] it was shown that such a "dual" execution allows the parties to perform a simple check to ensure that no additive errors were introduced in the multiplication gates. However, such check does not directly work over $\mathbb{Z}_{2^k}$, given that it relies on the fact that every non-zero element must be invertible, which only holds over fields.

In order to reduce the cheating success probability, we borrow the idea of [51] of working on the larger ring $\mathbb{Z}_{2^{k+s}}$. As we will show below, this ensures

that a similar check to that in [45] over fields can be carried out over $\mathbb{Z}_{2^{k+s}}$, ensuring no additive attacks over $\mathbb{Z}_{2^k}$ are carried out, except with probability at most 2^{-s}.

Protocol 2 Computing Arithmetic Circuits Over the Ring $\mathbb{Z}_{2^k}$

Inputs: Each party P_j ($j \in \{1, \ldots, n\}$) holds an input $x_j \in \mathbb{Z}_{2^k}^L$.

Auxiliary Input: The parties hold the description of an arithmetic circuit C over $\mathbb{Z}_{2^k}$ that computes f on inputs of length $M = L \cdot n$. Let N be the number of multiplication gates in C. In addition, the parties hold a parameter $s \in \mathbb{N}$.

1. *Secret sharing the inputs:*
   a) For each input x_j held by party P_j, party P_j represent it as an element of $\mathbb{Z}_{2^{k+s}}^L$ and sends x_j to $\mathcal{F}_{\mathsf{Input}}(k+s)$.
   b) Each party P_j records its vector of shares $(x_1^j, \ldots, x_M^j)$ of all inputs, as received from $\mathcal{F}_{\mathsf{Input}}(k+s)$. If a party received $\perp$ from $\mathcal{F}_{\mathsf{Input}}$, then it sends `abort` to the other parties and halts.
2. *Generate randomizing shares:* The parties call $\mathcal{F}_{\mathsf{Rand}}(k+s)$ to receive $[\![r]\!]_{k+s}$, where $r \in_R \mathbb{Z}_{2^{k+s}}$.
3. *Randomization of inputs:* For each input wire sharing $[\![v_m]\!]_{k+s}$ (where $m \in \{1, \ldots, M\}$) the parties call $\mathcal{F}_{\mathsf{Mult}}$ on $[\![r]\!]_{k+s}$ to receive $[\![r \cdot v_m]\!]_{k+s}$.
4. *Circuit emulation:* The parties traverse over the circuit in topological order. For each gate G_ℓ the parties work as follows:
   - *G_ℓ is an addition gate:* Given tuples $([\![x]\!]_{k+s}, [\![r \cdot x]\!]_{k+s})$ and $([\![y]\!]_{k+s}, [\![r \cdot y]\!]_{k+s})$ on the *left* and *right* input wires respectively, the parties locally compute $([\![x+y]\!]_{k+s}, [\![r \cdot (x+y)]\!]_{k+s})$.
   - *G_ℓ is a multiplication-by-a-constant gate:* Given a constant $a \in \mathbb{Z}_{2^k}$ and tuple $([\![x]\!]_{k+s}, [\![r \cdot x]\!]_{k+s})$ on the input wire, the parties locally compute $([\![a \cdot x]\!]_{k+s}, [\![r \cdot (a \cdot x)]\!]_{k+s})$.
   - *G_ℓ is a multiplication gate:* Given tuples $([\![x]\!]_{k+s}, [\![r \cdot x]\!]_{k+s})$ and $([\![y]\!]_{k+s}, [\![r \cdot y]\!]_{k+s})$ on the *left* and *right* input wires respectively:
     a) The parties call $\mathcal{F}_{\mathsf{Mult}}$ on $[\![x]\!]_{k+s}$ and $[\![y]\!]_{k+s}$ to receive $[\![x \cdot y]\!]_{k+s}$.
     b) The parties call $\mathcal{F}_{\mathsf{Mult}}$ on $[\![r \cdot x]\!]_{k+s}$ and $[\![y]\!]_{k+s}$ to receive $[\![r \cdot x \cdot y]\!]_{k+s}$.
5. *Verification stage:* Let $\{([\![z_i]\!]_{k+s}, [\![r \cdot z_i]\!]_{k+s})\}_{i=1}^N$ be the tuples on the output wires of all multiplication gates and let $\{[\![v_m]\!]_{k+s}, [\![r \cdot v_m]\!]_{k+s}\}_{m=1}^M$ be the tuples on the input wires of the circuit.

a) For $m = 1, \ldots, M$, the parties call $\mathcal{F}_{\mathsf{Rand}}(k+s)$ to receive $[\![\beta_m]\!]_{k+s}$.

b) For $i = 1, \ldots, N$, the parties call $\mathcal{F}_{\mathsf{Rand}}(k+s)$ to receive $[\![\alpha_i]\!]_{k+s}$.

c) *Compute linear combinations:*

i. The parties call $\mathcal{F}_{\mathsf{DotProduct}}$ on $([\![\alpha_1]\!]_{k+s}, \ldots, [\![\alpha_N]\!]_{k+s}, [\![\beta_1]\!]_{k+s}, \ldots, [\![\beta_M]\!]_{k+s})$ and $([\![r \cdot z_1]\!]_{k+s}, \ldots, [\![r, \cdot z_N]\!]_{k+s}, [\![r \cdot v_1]\!]_{k+s}, \ldots, [\![r \cdot v_M]\!]_{k+s})$ to obtain $[\![u]\!]_{k+s} = [\![\sum_{i=1}^{N} \alpha_i \cdot (r \cdot z_i) + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m)]\!]_{k+s}$.

ii. The parties call $\mathcal{F}_{\mathsf{DotProduct}}$ on $(\alpha_1, \ldots, \alpha_N, \beta_1, \ldots, \beta_M)$ and $([\![z_1]\!]_{k+s}, \ldots, [\![z_N]\!]_{k+s}, [\![v_1]\!]_{k+s}, \ldots, [\![v_M]\!]_{k+s})$ to obtain $[\![w]\!]_{k+s} = [\![\sum_{i=1}^{N} \alpha_i \cdot z_i + \sum_{m=1}^{M} \beta_m \cdot v_m]\!]_{k+s}$.

d) The parties run $\mathsf{open}([\![r]\!]_{k+s})$ to receive r.

e) Each party locally computes $[\![T]\!]_{k+s} = [\![u]\!]_{k+s} - r \cdot [\![w]\!]_{k+s}$.

f) The parties call $\mathcal{F}_{\mathsf{CheckZero}}(k+s)$ on $[\![T]\!]_{k+s}$. If $\mathcal{F}_{\mathsf{CheckZero}}(k+s)$ outputs `reject`, the parties output $\perp$ and abort. If it outputs `accept`, they proceed.

6. *Output reconstruction:* For each output wire of the circuit with $[\![v]\!]_{k+s}$, the parties locally convert to $[\![v]\!]_k$. Then, they run $v \bmod 2^k = \mathsf{open}([\![v]\!]_k, j)$, where P_j is the party whose output is on the wire. If P_j received $\perp$ from the open procedure, then it sends $\perp$ to the other parties, outputs $\perp$ and halts.

Output: If a party has not aborted, then it outputs the values received on its output wires.

At the core of the security of our protocol lies the following lemma, which shows that an additive attack that is non-zero modulo k in any multiplication gate leads to failure in the final check to zero, with overwhelming probability.

Lemma 1. *If $\mathcal{A}$ sends an additive value $d \not\equiv_k 0$ in any of the calls to $\mathcal{F}_{\mathsf{Mult}}$ in the execution of Protocol 2, then the value T computed in the verification stage of Step 5 equals 0 with probability* $2^{-s+\log(s+1)}$.

The proof of Lemma 1 is in Appendix 5.7.

The security of Protocol 2 now follows as Lemma 1 shows that additive errors that are non-zero modulo 2^k cannot take place without leading to abort. However, one non-trivial issue lies in handling additive attacks that are zero modulo 2^k, but not modulo 2^{k+s}, as these do not affect correctness but may lead to selective failure attacks, in which an abort signal can be generated depending on the inputs from honest parties. Our protocol deals with this

potential attack by using secret coefficients for the random linear combination taken in the verification step. If we take public coefficients, as done in [45], the following attack can be carried out.

Assume that the adversary has attacked exactly one gate, indexed by i_0, in the following way. When multiplying x_{i_0} with y_{i_0}, the adversary acted honestly, but when multiplying $r \cdot x_{i_0}$ with y_{i_0}, it added the value d_{i_0}. Thus, on the output wire, the parties hold a sharing of the pair $(x_{i_0} \cdot y_{i_0}, r \cdot x_{i_0} \cdot y_{i_0} + d_{i_0})$. Now, assume that this wire enters another multiplication gate, indexed by j_0 with input shares on the second wire being $(w_{j_0}, r \cdot w_{j_0})$ and that the output of this second gate is an output wire of the circuit. Thus, on the output of this gate, the parties will hold the sharing $(x_{i_0} \cdot y_{i_0} \cdot w_{j_0}, (r \cdot x_{i_0} \cdot y_{i_0} + d_{i_0})w_{j_0})$ (assuming the adversary does not attack this gate as well). In this case, we have that $T = \alpha_{i_0} \cdot d_{i_0} + \alpha_{j_0} \cdot (d_{i_0} \cdot w_{j_0}) = d_{i_0}(\alpha_{i_0} + \alpha_{j_0} \cdot w_{j_0})$. Now, if $d_{i_0} = 2^{k+s-1}$ then it follows that $T \equiv_{k+s} 0$ if and only if $\alpha_{i_0} + \alpha_{j_0} \cdot w_{j_0}$ is even.

The attack presented above does not change the k lower bits of the values on the wires, and thus has no effect on the correctness of the output. However, if α_{i_0} and α_{j_0} are public and known to the adversary, then by $\mathcal{F}_{\mathsf{CheckZero}}$'s ouptut the adversary may be able to learn whether w_{j_0} is even or not. In contrast, when α_{i_0} and α_{j_0} are kept secret, learning whether $\alpha_{i_0} + \alpha_{j_0} \cdot w_{j_0}$ is even or odd does not reveal any information about w_{j_0} since it is now perfectly masked by α_{i_0} and α_{j_0}. Therefore, to prevent this type of attack, we are forced to use random secrets for our random linear combination. Here is where the functionality $\mathcal{F}_{\mathsf{DotProduct}}$ becomes handy, as it allows to compute the sum of products of sharings in an efficient way which is exactly what we need to compute $\sum_{i=1}^{N} [\![\alpha_i]\!] \cdot [\![z_i]\!]$.

We state the security of our protocol below. A full simulation-based proof appears in the full version of this work.

Theorem 1. *Let f be an n-party functionality over $\mathbb{Z}_{2^k}$ and let s be a statistical security parameter. Then, Protocol 2 securely computes f with abort in the $(\mathcal{F}_{\mathsf{Input}}, \mathcal{F}_{\mathsf{Mult}}, \mathcal{F}_{\mathsf{Coin}}, \mathcal{F}_{\mathsf{Rand}}, \mathcal{F}_{\mathsf{CheckZero}})$-hybrid model with statistical error $2^{-s+\log(s+1)}$, in the presence of a malicious adversary controlling $t < \frac{n}{2}$ parties.*

Concrete efficiency

We now analyze the performance of the protocol. Recall that M is the number of inputs and N is the number of multiplication gates in the circuit. We denote by O the number of output wires of the circuit, and for a given functionality $\mathcal{F}_*(\ell)$, we denote by $\mathcal{C}_*(\ell)$ the communication cost (in bits) of calling this primitive.

For each input wire, we have one call to $\mathcal{F}_{\mathsf{Input}}(k+s)$, which is translated into one call to $\mathcal{F}_{\mathsf{Rand}}(k+s)$, one call to $\mathsf{open}([\![r]\!]_{k+s}, i)$ and one element in $\mathbb{Z}_{2^{k+s}}$ that is sent by some party P_i to the other parties. In addition,

there is one call to $\mathcal{F}_{\mathsf{Mult}}(k+s)$ to randomize each input. This adds up to $M \cdot (2 \cdot \mathcal{C}_{\mathsf{rand}}(k+s) + \mathcal{C}_{\mathsf{open(i)}}(k+s) + (k+s))$.

For each multiplication gate, we call $\mathcal{F}_{\mathsf{Mult}}(k+s)$ twice. Then, in the verification step, $\mathcal{F}_{\mathsf{Rand}}(k+s)$ is called for each input wire and multiplication gate. This adds $N \cdot (\mathcal{C}_{\mathsf{rand}} + 2 \cdot \mathcal{C}_{\mathsf{mult}}(k+s))$. The remaining of the verification step consists of two calls to $\mathcal{F}_{\mathsf{DotProduct}}(k+s)$, one call to $\mathsf{open}(\llbracket r \rrbracket_{k+s})$ and one call to $\mathcal{F}_{\mathsf{CheckZero}}(k+s)$. Recall that we assume that the protocol realizing $\mathcal{F}_{\mathsf{DotProduct}}(k+s)$ has the same communication complexity as $\mathcal{F}_{\mathsf{Mult}}(k+s)$, so this adds up to $2 \cdot \mathcal{C}_{\mathsf{mult}}(k+s) + \mathcal{C}_{\mathsf{open(i)}}(k+s) + \mathcal{C}_{\mathsf{CheckZero}}(k+s)$. However, as these are small constants which do not depend on the size of the circuit, we exclude them from the final count. In the output reconstruction step, for each output wire, there is one call to $\mathsf{open}(\llbracket v \rrbracket_k, i)$.

We thus have that the cost of the protocol is

$$M \cdot \big(2 \cdot \mathcal{C}_{\mathsf{rand}}(k+s) + \mathcal{C}_{\mathsf{mult}}(k+s) + \mathcal{C}_{\mathsf{open(i)}}(k+s) + (k+s)\big) \\ + N \cdot (\mathcal{C}_{\mathsf{rand}}(k+s) + 2 \cdot \mathcal{C}_{\mathsf{mult}}(k+s)) + O \cdot \mathcal{C}_{\mathsf{open(i)}}(k).$$

For circuits where $N \gg M, O$ (i.e., there are much more multiplication gates than input and output wires), this is translated to $N \cdot (\mathcal{C}_{\mathsf{rand}}(k+s) + 2 \cdot \mathcal{C}_{\mathsf{mult}}(k+s))$. Notice that for some instantiations, like the replicated secret sharing based one from Appendix 5.7, $\mathcal{F}_{\mathsf{Rand}}$ is "for free" in the sense that it can be implemented efficiently by relying on a computational assumption, e.g., PRGs with correlated keys.

3.5 Instantiation for n parties

In this section, we present our instantiation based on Shamir's secret sharing over rings, using the techniques from [5]. This technique works for any number of parties, although for 3 parties one can obtain more efficient solutions, such as the one we describe in Appendix 5.7 that uses replicated secret sharing. Over finite fields, Shamir's scheme requires a distinct evaluation point for each player, and one more for the secret. This is usually not a problem if the size of the field is not too small. However, over commutative rings R the condition on the sequence of evaluation points $\alpha_0, \ldots, \alpha_n \in R$ is that the pairwise difference $\alpha_i - \alpha_j$ is invertible for each pair of indices $i \neq j$. For our ring of interest $\mathbb{Z}_{2^\ell}$, the largest such sequence the ring admits is only of length 2 (e.g. $(\alpha_0, \alpha_1) = (0, 1)$).

The solution from [5] is to embed inputs from $\mathbb{Z}_{2^\ell}$ into a large enough Galois ring R that has $\mathbb{Z}_{2^\ell}$ as a subring. This ring is of the form $R = \mathbb{Z}_{2^\ell}[X]/(h(X))$, where $h(X)$ is a monic polynomial of degree $d = \lceil \log_2 n \rceil$ such that $h(X) \bmod 2 \in \mathbb{F}_2[X]$ is irreducible. Elements of R thus correspond uniquely to polynomials with coefficients in $\mathbb{Z}_{2^\ell}$ that are of degree at most $d-1$. Note the similarity between the Galois ring and finite field extensions of $\mathbb{F}_2$:

elements of the finite field $\mathbb{F}_{2^d}$ correspond uniquely to polynomials of at most degree $d-1$ with coefficients in $\mathbb{F}_2$.

There is a ring homomorphism $\pi : R \to \mathbb{Z}_{2^\ell}$ that sends $a_0 + a_1 X + \cdots + a_{d-1}X^{d-1} \in R$ to the free coefficient a_0, which we shall use later on.[2] For more relevant structural properties of Galois rings, see [5].

We adopt the above-mentioned version of Shamir's scheme over R, but restrict the secret space to $\mathbb{Z}_{2^\ell}$. The share space will be equal to R. Let $1 \leq \tau \leq n$ be the privacy parameter of the scheme. Then, the set of *correct* share vectors is

$$C_\tau = \left\{ (f(\alpha_1), \ldots, f(\alpha_n)) \in R^n \;\middle|\; \begin{array}{c} f \in R[X],\ \deg(f) \leq \tau, \\ \text{and } f(\alpha_0) \in \mathbb{Z}_{2^\ell} \subset R \end{array} \right\}. \tag{3.1}$$

With the restriction that the secret is in $\mathbb{Z}_{2^\ell}$, we have that C_τ is an $\mathbb{Z}_{2^\ell}$-module, i.e. the secret-sharing scheme is $\mathbb{Z}_{2^\ell}$-linear. Since it is based on polynomial interpolation, the properties from 3.2 can be easily seen to hold. This includes division by 2 if all the shares are even.

In this section, we denote a sharing under C_τ as $[\![x]\!]_\tau = (x_1, \ldots, x_n)$. We call τ the *degree* of the sharing. The reason we are explicit about τ is that we will use sharings of two different degrees. This stems from the critical property of this secret-sharing scheme that enables us to evaluate arithmetic circuits: this secret-sharing scheme is *multiplicative*. This means there is a $\mathbb{Z}_{2^\ell}$-linear map $R^n \to \mathbb{Z}_{2^\ell}$ that for sharings $[\![x]\!]_\tau, [\![y]\!]_\tau$ sends $(x_1y_1, \ldots, x_ny_n) \mapsto x \cdot y$.

Put differently, $(x_1y_1, \ldots, x_ny_n) \in C_{2\tau}$ is a degree-2τ sharing with secret $x \cdot y$. We denote it $[\![x \cdot y]\!]_{(2\tau)} = (x_1y_1, \ldots, x_ny_n)$ — in particular note the parenthesized subscript refers to the degree of the sharing, as opposed to the modulus. Note that $C_i \subseteq C_j$ for $0 < i < j$; in particular every degree-2τ sharing is also a sharing of degree $n-1$. A sharing of degree $n-1$ is related to additive secret sharing, where the secret equals the sum of the shares $x = \sum_i x_i$. The difference is that here there are constants, i.e. we may write $x = \sum_i \lambda_i x_i$, for $\lambda_1, \ldots, \lambda_n \in R$. We shall make use of this in our multiplication protocol, ensuring that parties only need to communicate an element of $\mathbb{Z}_{2^\ell}$ instead of an element of R. However, note that $[\![\cdot]\!]_{(2\tau)}$ does not meet the definition of a secret-sharing scheme in Section 3.2, in particular because the corrupted parties shares are not well defined and cannot be computed from the honest parties' shares.

Generating Randomness

We efficiently realize $\mathcal{F}_{\mathsf{Rand}}$ by letting each player P_i sample and secret-share a random element s_i, and then multiplying the resulting vector of n random

[2] Technically, an element of R is a residue class modulo the ideal $(h(X))$, but we omit this for simplicity of notation.

elements with a particular[3] Vandermonde matrix [58].[4] Of the resulting vector, τ entries are discarded to ensure the adversary has zero information about the remaining ones. Thus, $n - \tau$ random elements are outputted, resulting in an amortized communication cost of $O(n)$ ring elements per element. A priori the adversary can cause the sharings to be incorrect; this is remedied with Protocol 4 by opening a random linear combination of the sharings and verifying the result.

Since our secret-sharing scheme $[\![\cdot]\!]_\tau$ is $\mathbb{Z}_{2^\ell}$-linear, we would like to choose our matrix with entries in $\mathbb{Z}_{2^\ell}$. Unfortunately, the Vandermonde matrix we need does not exist over $\mathbb{Z}_{2^\ell}$, for the same reason secret sharing does not work. However, the secret-sharing scheme which consists of d parallel sharings of $[\![\cdot]\!]_\tau$ be interpreted as an R-linear secret-sharing scheme [5, 37]. This secret-sharing scheme, which we denote as $\langle\cdot\rangle$, has share space S^d (since the scheme is identical to sharing d independent secrets in S in parallel using $[\![\cdot]\!]_\tau$), and secret space R^d. The scheme is R-linear because the module of share vectors, which is $(C_\tau)^d$, is an R-module via the tensor product $(C_\tau)^d \cong C_\tau \otimes_S S^d \cong C_\tau \otimes_S R$. In practice, a single secret-shared element $\langle x\rangle$ may be interpreted as a secret-shared column vector $([\![x_1]\!]_\tau, \ldots, [\![x_d]\!]_\tau)^T$. To compute the action of an element $r \in R$ on $\langle x\rangle$ in this representation, we first need to fix a basis of R over S. Recall $R = \mathbb{Z}_{2^\ell}[X]/(h(X))$, so we may pick the canonical basis $1, X, \ldots, X^{d-1} \in R$. This allows us to represent an element $a \in R$ as a column vector $(a_0, \ldots, a_{d-1})^T \in S^d$, i.e. explicitly: $a = a_0 + a_1 X + \cdots + a_{d-1}X^{d-1}$. Multiplication by $r \in R$ is an S-linear map of vectors $S^d \to S^d$, i.e. it can be represented as a $d \times d$ matrix M_r with entries in S. The product $r\langle x\rangle = \langle rx\rangle$ is then equal to $M_r([\![x_1]\!]_\tau, \ldots, [\![x_d]\!]_\tau)^T$. If a single party P has a vector of shares $(s_1, \ldots, s_d) \in R$ for $\langle x\rangle = ([\![x_1]\!]_\tau, \ldots, [\![x_d]\!]_\tau)^T$, then $M_r(s_1, \ldots, s_d)^T$ is their vector of shares corresponding to $\langle rx\rangle$.

In our protocol, the parties compute $(\langle r_1\rangle, \ldots, \langle r_{n-\tau}\rangle)^T = A(\langle s_1\rangle, \ldots, \langle s_n\rangle)^T$, where A has entries in R. This can be computed by writing out the R-linear combinations $\langle r_i\rangle = \sum_{k=1}^n a_{ik}\langle s_k\rangle = \sum_{k=1}^n M_{a_{ik}}\langle s_k\rangle$, with $\langle s_k\rangle = ([\![s_{k1}]\!]_\tau, [\![s_{kd}]\!]_\tau)^\intercal$. Fix a sequence $\beta_1, \ldots, \beta_n \in R$ such that for each pair of indices $i \neq j$ we have that $\beta_i - \beta_j$ is invertible.[5] We let A be the $(n-\tau) \times n$ matrix such that the j-th column is $(1, \beta_j, \beta_j^2, \ldots, \beta_j^{n-\tau-1})^T$. This matrix is hyperinvertible, i.e. any square submatrix is invertible [5].

[3]Over fields this can be a general Vandermonde matrix, but this is not sufficient over R.

[4]In general, any R-linear code with good distance and dimension suffices to get $O(n)$ complexity in the protocol, but the Vandermonde construction is optimal.

[5]We may just use $(\beta_1, \ldots, \beta_n) = (\alpha_1, \ldots, \alpha_n)$.

Protocol 3 Generating random sharings of $[\![\cdot]\!]_\tau$

1. Each party P_i samples an element $s_i \leftarrow (\mathbb{Z}_{2^\ell})^d$ and secret-shares it as $\langle s_i \rangle$ among all parties.
2. The parties locally compute the linear matrix-vector product to obtain $(\langle r_1 \rangle, \ldots, \langle r_{n-\tau} \rangle)^T := A(\langle s_1 \rangle, \ldots, \langle s_n \rangle)^T$.
3. The parties execute Protocol 4 $\lceil \kappa/d \rceil$ times in parallel on $\langle r_1 \rangle, \ldots, \langle r_{n-\tau} \rangle$If any execution fails, they abort. Otherwise, for each $j = 1, \ldots, n-\tau$ they interpret $\langle r_j \rangle = ([\![r_{j1}]\!]_\tau, \ldots, [\![r_{jd}]\!]_\tau)$ and output $[\![r_{11}]\!]_\tau, \ldots, [\![r_{1d}]\!]_\tau, [\![r_{21}]\!]_\tau, \ldots, [\![r_{(n-\tau)d}]\!]_\tau$.

Lemma 2. *Protocol 3 securely computes $(n-\tau)d$ parallel invocations of $\mathcal{F}_{\mathsf{Rand}}$ for $[\![\cdot]\!]_\tau$ with statistical error of at most $2^{-\kappa}$ in the presence of a malicious adversary controlling $t < n/2$ parties.*

The proof is in Appendix 5.7

Checking Correctness of Sharings

We check whether sharings are correct by taking a random linear combination of the sharings, masking it with a random sharing, and opening the result to all parties.

This protocol does not securely compute an ideal functionality, because privacy is not preserved if the sharings are incorrect. The way we use it this does not matter, since we only verify correctness of sharings of random elements.

Protocol 4 Checking correctness of sharings of $\langle \cdot \rangle$

Input: possibly incorrect sharings $\langle x_1 \rangle, \ldots, \langle x_N \rangle$, and a possibly incorrect sharing $\langle r \rangle \leftarrow (\mathbb{Z}_{2^\ell})^d$ of a random element.

1. The parties call $\mathcal{F}_{\mathsf{Coin}}$ N times to get $a_1, \ldots, a_N \leftarrow (\mathbb{Z}_{2^\ell})^d$.
2. The parties compute $\langle u \rangle := a_1 \langle x_1 \rangle + \cdots + a_N \langle x_N \rangle + \langle r \rangle$.
3. The parties run $\mathsf{open}(\langle u \rangle)$. If it returns $\perp$, output $\perp$. Else, output $\mathsf{correct}$.

Lemma 3. *If at least one of the input sharings $\langle x_1 \rangle, \ldots, \langle x_N \rangle$ is incorrect, Protocol 4 outputs* $\mathsf{correct}$ *with probability at most $\frac{1}{2^d}$.*

To show correctness, we use the following consequence from [5, Lemma 3].

Lemma 4. *Let $C \subseteq R^n$ be a free R-module. Then for all $x \notin C$ and $u \in R^n$, we have that*

$$\Pr_{r \leftarrow R}[rx + u \in C] \leq \frac{1}{2^d}$$

where r is chosen uniformly at random from R.

Proof of Lemma 3. Let C denote the R-module of correct share vectors (such as in (3.1)). One of the input sharings is incorrect; without loss of generality assume it is $\langle x_1 \rangle$. The protocol $\mathsf{open}(\langle u \rangle)$ returns a value not equal to $\perp$ if and only if $\langle u \rangle = a_1 \langle x_1 \rangle + (a_2 \langle x_2 \rangle + \cdots + a_n \langle x_n \rangle + \langle r \rangle)$ is in C. By Lemma 4 this probability is bounded by $1/2$, since a_1 was chosen uniformly at random. Since $\langle u \rangle$ is masked with $\langle r \rangle$, the protocol is private. ☐

Secure Multiplication up to Additive Attacks

Multiplication follows the outline of the passively secure protocol of [58]. The protocol begins with an offline phase, where *random double sharings* are produced, i.e. a pair of sharings $(\llbracket r \rrbracket_\tau, \llbracket r \rrbracket_{(2\tau)})$ of the same uniformly random element r shared using polynomials of degree τ and degree 2τ, respectively.

We denote a double sharing as $\llbracket r \rrbracket_{(\tau,2\tau)} := ((r_1, r'_1), \ldots, (r_n, r'_n))$. It is a $\mathbb{Z}_{2^\ell}$-linear secret-sharing scheme with secret space $\mathbb{Z}_{2^\ell}$ and share space $R \oplus R$. The set of correct share vectors is the $\mathbb{Z}_{2^\ell}$-module

$$\left\{ ((f(\alpha_1), g(\alpha_1)), \ldots, (f(\alpha_n), g(\alpha_n))) \;\middle|\; \begin{array}{c} f, g \in R[X], \\ f(\alpha_0) = g(\alpha_0) \in \mathbb{Z}_{2^\ell}, \\ \deg(f) \leq \tau,\ \deg(g) \leq 2\tau \end{array} \right\}.$$

Secret-sharing an element r under $\llbracket \cdot \rrbracket_{(\tau,2\tau)}$ involves selecting two uniformly random polynomials of degrees at most τ and 2τ respectively.

To generate sharings in $\llbracket \cdot \rrbracket_{(\tau,2\tau)}$, we essentially use Protocol 3. However, this protocol does not securely realize $\mathcal{F}_{\mathsf{Rand}}$, since in Lemma 2 we use the fact that the simulator can compute the corrupted parties' shares from the honest parties' shares, which is not the case for the degree-2τ part (hence why $\llbracket \cdot \rrbracket_{(2\tau)}$, therefore also $\llbracket \cdot \rrbracket_{(\tau,2\tau)}$, does not meet the definition of a secret-sharing scheme in Section 3.2). This will only lead to an additive attack in the online phase, which is why we can still use the protocol.

Protocol 5 Secure multiplication up to an additive attack

Inputs: Parties hold correct sharings $\llbracket x \rrbracket_\tau$, $\llbracket y \rrbracket_\tau$

Offline phase: The parties execute Protocol 3 for $\llbracket \cdot \rrbracket_{(\tau,2\tau)}$ instead of $\llbracket \cdot \rrbracket_\tau$. They only check correctness for the $\llbracket \cdot \rrbracket_\tau$ part, and not for the $\llbracket \cdot \rrbracket_{(2\tau)}$ part. They obtain a random double sharing $(\llbracket r \rrbracket_\tau, \llbracket r \rrbracket_{(2\tau)})$.

Online phase:

1. The parties locally calculate $[\![\delta]\!]_{(2\tau)} := [\![x]\!]_\tau \cdot [\![y]\!]_\tau - [\![r]\!]_{(2\tau)}$.
2. Each P_i for $i = 1, \dots 2\tau + 1$ sends $u_i := \pi(\lambda_i \delta_i)$ to P_1 (recall $\pi(a_0 + a_1 X + \cdots + a_{d-1}X^{d-1}) = a_0 \in \mathbb{Z}_{2^\ell}$, and the λ_i are constants such that $\sum_{i=1}^n \lambda_i \delta_i = \delta$)
3. P_1 can now reconstruct δ as $\delta = \sum_{i=1}^n u_i$.
4. P_1 broadcasts δ.
5. The parties locally compute $[\![x \cdot y]\!]_\tau = [\![r]\!]_\tau + \delta$.

The reason each party sends u_i instead of δ_i to P_1 is two-fold. It saves bandwidth, since only an element of $\mathbb{Z}_{2^\ell}$ needs to be communicated instead of an element of R. More importantly though, if the inputs $[\![x]\!]_\tau, [\![y]\!]_\tau$ are not guaranteed to be correct, then sending full shares δ_i can compromise privacy.

Note that it is important that the random double sharing $[\![r]\!]_{(\tau,2\tau)}$ is guaranteed to be correct. I.e., the shares are degree τ and 2τ respectively.

Lemma 5. *Protocol 5 securely computes $\mathcal{F}_{\mathsf{Mult}}$ with statistical error $\leq 2^{-\kappa}$ in the $\mathcal{F}_{\mathsf{Rand}}$-hybrid model in the presence of a malicious adversary controlling $t < n/2$ parties.*

The proof appears in Section 5.7.

When evaluating a circuit gate-by-gate using Protocol 5, we consider an optimization in which we do not need to execute the broadcast (which might be expensive) for each multiplication, but instead they will perform a broadcast just before opening the values. In the multiplication protocol, P_1 will just send a value (not guaranteed to be the same) to all other parties. Each party P_i keeps track of a hash value h_i of all received values in step 4 of the protocol far. Before opening their outputs, each party P_i sends its hash h_i to all other parties. If any party detects a mismatch, they abort. Note that security up to additive attack is guaranteed only after this procedure succeeds, which is executed before opening the output.

In doing so, we lose the invariant that all secret-shared values are guaranteed to be correct. In other scenarios, as for example the $t < n/3$ setting, this completely breaks the security of the protocol as shown in [87]. However, this is not a problem in our case since the degree-2τ sharings have no redundancy in them. As shown in [87], this is enough to guarantee the security of the protocol with the deferred check, and the reason is essentially that the shares that the potentially corrupt party P_1 receives are now uniformly random and independent of each other.

3.6 Implementation and Evaluation

We report in the following section on an implementation of both the Shamir based instantiation, as well as the 3-party instantiation based on replicated secret-sharing.

Implementation Details

We implement both protocols in C++ and rely on `uint64_t` and `unsigned int128`[6] types for arithmetic over $\mathbb{Z}_{2^\ell}$, where the former is used when $\ell = 64$ and the latter when $\ell = 128$. Notice that this choice allows us to investigate two sets of parameters: $\ell = 64$ can be viewed as 32 bit computation with 32 bits of statistical security, while $\ell = 128$ gives us 64 bits of computation with 64 bits of statistical security. We rely on libsodium for hashing and the PRG we use is based on AES.

For the Galois-ring variant our implementation uses the ring $R = \mathbb{Z}_{2^\ell}[X]/(h(x))$ with $h(X) = X^4 + X + 1$ and denote this by $GR(2^\ell, d = 4)$. This ring supports $2^4 - 1 = 15$ parties and the act of hard-coding the irreducible polynomial allows us to implement multiplication and division in the ring using lookup tables. It is worth remarking that operations in $GR(2^\ell, d)$ are more expensive than certain prime fields (in particular, Mersenne primes as the ones used in [45]). Concretely, a multiplication in $GR(2^{64}, 4)$ require 20 `uint64_t` multiplications and 18 additions, while a multiplication in $\mathbb{Z}_{2^{64}}$ require only a couple of `uint64_t` multiplications as well as a few bitwise operations. so while some MPC primitives in $\mathbb{Z}_{2^\ell}$ may be cheaper (for example, masking a value in $\mathbb{Z}_{2^\ell}$ is cheaper), this gain in efficiency is greatly reduced by the complexity of operating in the Galois-Ring.

Experimental setup. We run our experiments on AWS `c5.9xlarge` machines, which have 36 virtual cores, 72gb of memory and a 10Gpbs network. We utilize 3 separate machines and so for experiments with $n > 3$, some parties run on the same machine. However, the load on each machine is distributed evenly (e.g., with 5 parties, the first two machines each run 2 parties each while the last run only 1 party).

Experiments

Our experiments comprises two points of comparison:

First we compare our Shamir based instantiation (cf. Section 3.5) against the field protocol of [45]. For this, we use the implementation at [4]. We perform the same benchmarks as reported on in [45]; that is, circuits of varying depth with a fixed number of parties. Each experiment is repeated for n set to

[6]This type is a GCC extension, cf. `https://gcc.gnu.org/onlinedocs/gcc/_005f_005fint128.html`

3, 5, 7 and 9. The main goal here is to understand the overhead of working with $GR(2^\ell, d)$ as opposed to working over $\mathbb{Z}_p$. As [4] supports different choices of the prime p we set p to be a 61-bit Mersenne prime, as this is the most efficient field that also allows for a reasonable expressive computations.

Our second set of experiments will compare our replicated instantiation (cf. Appendix 5.7) against the protocols for computation over rings presented in [67]. In these experiments we measure the throughput of multiplications in our protocol; that is, how many multiplications our protocol can compute per second. Since we do not have access to the implementations of [67], we opt instead to use the experimental setup as theirs, in order to obtain a fair comparison. We report here on benchmarks run in a LAN setting.

While the protocol of [45] is the natural choice for comparing our n-party instantiation, a number of efficient specialized 3 party protocols exist which we briefly mention here. We choose the protocols of [67] for comparison as their experiments and setup is straightforward to replicate with our protocol, thus allowing us to make a fair comparison. Concurrently with [67], several other proposals for 3 party protocols have been published, such as [41] or [124]. However, no public implementation exist for these works, and the nature of the experiments they perform makes it very hard to perform a fair comparison (as we do later with the results from [67]). More precisely, both [41] and [124] evaluate their protocols relative to an implementation of ABY3 [117] that was also implemented by the authors themselves (as no public implementation of ABY3 was available at that time).

While [124] have better amortized communication cost, we estimate that their *concrete* running time (when considering end-to-end times, as we do in this work) will be worse. We base this conjecture on the fact that [124] uses the interpolation based check from [30]. For the case of fields, this check was shown in [32] to take several seconds in order to check 1 million multiplications (which is the benchmark we use). Running the same check, but over a ring, requires computation over a fairly large extension of $\mathbb{Z}_{2^k}$, which we have no reason to expect would be significantly faster than the field based check. Concluding, we would not be surprised if [124] is faster *in the online phase*; however, preprocessing the triples needed to get this would be much slower than our protocol. We stress that our protocol (for the 3 party case) has *no preprocessing*, so we expect our protocol to perform much better when measuring end-to-end times. We elaborate a bit more on the cost of the kind of check used in [124] later, when we discuss [32].

Results: Shamir instantiation

The results of our experiments can be seen in Table 3.1. Across the board, we see that preprocessing is more expensive in our protocol than in [45]. However, the overhead is in lines with the observation made above that operating in $GR(2^\ell, d)$ is about 4 times as expensive than in $\mathbb{Z}_p$ when $\ell = 64$ and p is a

61-bit Mersenne prime. This is in particular true when the number of parties is small as here local computation is the dominant factor. Moving to a larger number of parties, the overhead decreases, which we attribute to differences in the efficiency of the communication layer between our protocol and the one in [45].

Interestingly, we see for a lower number of parties but for very deep circuits, that our protocol performs better in the online phase. E.g., [45] takes 7.3 seconds, while both of our version is below 4.5 seconds. One reason for this could again be differences in the communication layer (since both our protocols communicate roughly the same amount of information due to the fact that we only need to send a $\mathbb{Z}_{2^\ell}$ element during reconstruction). However, our protocol is again less efficient when the number of parties increase, which would be due to the fact that the king needs to send more data during reconstruction, as well as the increased cost of the broadcast when more parties are involved. We remark that it is possible to distribute the broadcast load of the king among several parties, which may close the gap to some extent.

Finally, we see an expected overhead of roughly $\times 2$ between $\ell = 64$ and $\ell = 128$ (consider the depth 20 row in Table 3.1, as this is the setting where differences in local computation is most prominent). This more or less confirms the intuition that an operation in $\mathbb{Z}_{2^{128}}$ is around 2-3 times as expensive compared to an operation in $\mathbb{Z}_{2^{64}}$.[7]

Depth	**Protocol**	**3**	**5**	**7**	**9**
20	Ours $\ell = 64$	1.56 / 0.18	2.12 / 0.28	2.46 / 0.37	2.70 / 0.47
	Ours $\ell = 128$	2.79 / 0.52	4.28 / 0.74	4.73 / 0.91	5.10 / 1.11
	[45]	0.43 / 0.18	0.63 / 0.22	0.93 / 0.45	1.03 / 0.28
100	Ours $\ell = 64$	1.50 / 0.23	1.97 / 0.30	2.30 / 0.37	2.76 / 0.41
	Ours $\ell = 128$	2.80 / 0.51	3.78 / 0.61	4.15 / 0.77	5.02 / 0.95
	[45]	0.42 / 0.42	0.64 / 0.22	0.90 / 0.52	1.04 / 1.27
1,000	Ours $\ell = 64$	1.58 / 0.67	1.95 / 1.08	2.23 / 1.43	2.62 / 1.84
	Ours $\ell = 128$	2.80 / 1.23	3.68 / 1.81	4.23 / 2.08	5.03 / 2.47
	[45]	0.41 / 0.96	0.63 / 0.68	0.89 / 0.95	1.05 / 1.17
10,000	Ours $\ell = 64$	1.50 / 3.85	2.01 / 8.55	2.41 / 13.41	2.65 / 16.76
	Ours $\ell = 128$	2.81 / 4.43	3.71 / 8.07	4.38 / 13.31	5.03 / 16.43
	[45]	0.38 / 7.30	0.61 / 7.32	0.89 / 8.40	1.05 / 12.88

Table 3.1: LAN running times in seconds for circuits with 10^6 multiplications, different depth and for varying number of parties, evaluated using Shamir SS-based MPC. Each value is a tuple a/b where a is the preprocessing time (which is dominated by the double-share generation) and b is the time it takes to evaluate the circuit.

[7] Indeed, while a multiplication in $\mathbb{Z}_{2^{64}}$ is one unsigned 64-bit multiplication, a multiplication on 128-bit types compile to three $\mathbb{Z}_{2^{64}}$ multiplications. That the overhead is less than $3x$ can be attributed to the compiler being able to easier vectorize 64-bit multiplications in the $\mathbb{Z}_{2^{128}}$ case.

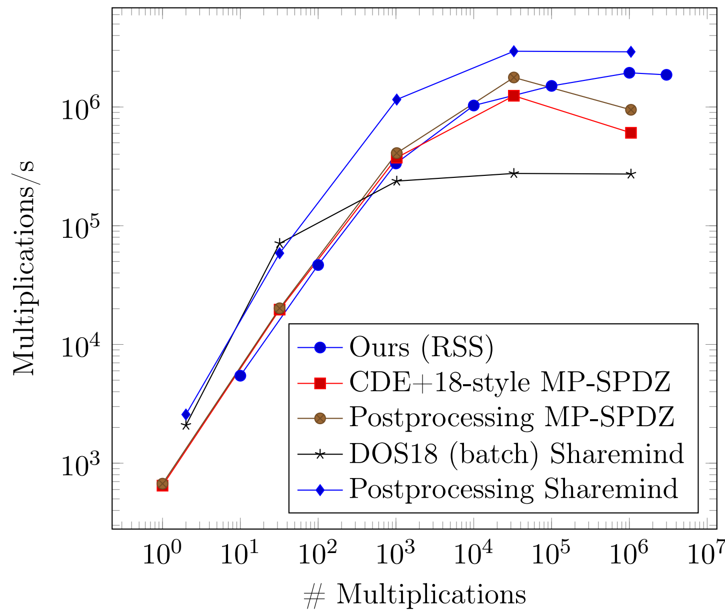


Figure 3.1: LAN throughput for replicated secret-sharing with 3 parties.

Comparing our instantiation with [45]. It is worth remarking that, for more elaborate protocols such as bit decompositions or truncations, operating over a prime field requires additional space for masking. For example, if we require 40 bits of security for masking, the 61-bit Mersenne prime only leaves room for ≈ 21 bits of computation. For these applications therefor, it is more reasonable to compare the numbers for [45] in Table 3.1 with our protocol with $\ell = 64$ (since $\mathbb{Z}_{2^k}$ does not require this extra space, $\ell = 64$ gives us 24 bits of computation at 40 bits of security). Alternatively, one could move to a 89-bit Mersenne or 127-bit Mersenne prime (allowing 49 and 87 bits of computation with 40 bits of security); however efficient multiplication in these fields require multiplication of essentially 128-bit integers without overflow, bringing it closer to operating in $GR(2^{128}, d)$.

Results: Replicated instantiation

We also compare our replicated instantiation with the protocols of [67], results of which can be seen in Figure 3.1 and Figure 3.2.[8] As we do not have access to the code of all the protocols considered in [67], we run our protocol in the same setup. With the exception of the Sharemind postprocessing protocol, we observe that we outperform all protocols of [67]. We may attribute this to the fact that both Sharemind and MP-SPDZ are more mature codebases and thus it is likely that a greater effort has been put into optimizations.

[8] We thank the authors of [67] for giving us the tikz code of their graph.

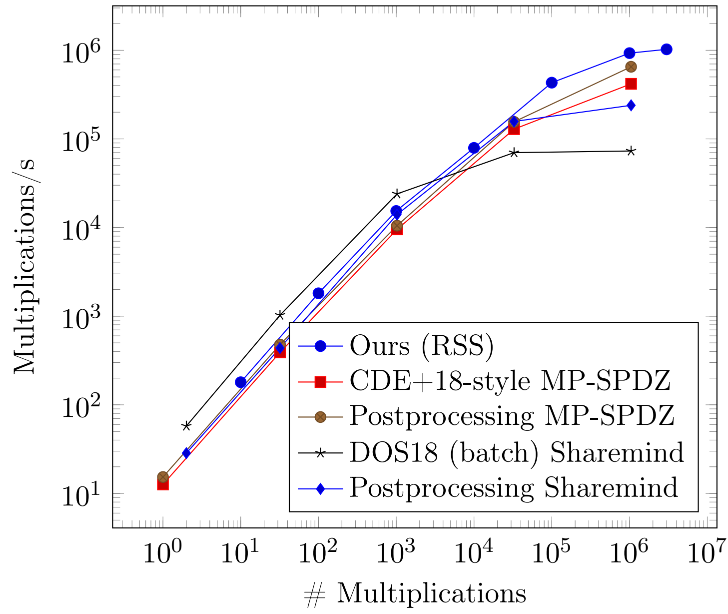


Figure 3.2: WAN throughput for replicated secret-sharing with 3 parties.

However, when we consider our protocol running in a WAN, we see that we outperform all protocols in [67]. This concurs with the fact that our protocol only needs to send 2 ring elements per multiplication, while the postprocessing protocols of [67] needs to send 3.

Chapter 4

Secure Inference of Quantized Neural Networks

Anders Dalskov[†], Daniel Escudero[†], Marcel Keller[‡]

† Aarhus University, Denmark
‡ CSIRO's Data61, Australia

Abstract. Image classification using Deep Neural Networks that preserve the privacy of both the input image and the model being used, has received considerable attention in the last couple of years. Recent work in this area have shown that it is possible to perform image classification with realistically sized networks using e.g., Garbled Circuits (XONN, USENIX '19) or MPC (CrypTFlow, Eprint '19). These, and other prior work, however, require models to be either trained in a specific way or postprocessed in order to be evaluated securely.

We contribute to this line of research by showing that this postprocessing can be handled by *standard* Machine Learning frameworks. More precisely, we show that *quantization* as present in Tensorflow and other frameworks suffices to obtain models that can be evaluated directly and *as-is* in standard off-the-shelve MPC. We implement secure inference of these quantized models in MP-SPDZ, and the generality of our technique means we can demonstrate benchmarks for a wide variety of threat models, something that has not been done before. In particular, we provide a comprehensive comparison between running secure inference of large ImageNet models with active and passive security, as well as honest and dishonest majority. The most efficient inference can be performed using a passive honest majority protocol which takes between 0.18 and 13.1 seconds, depending on the size of the model; for active security and an honest majority, inference is possible between 5.3 and 96.2 seconds.

4.1 Introduction

Machine Learning (ML) models are becoming more relevant in our day-to-day lives due to their ability to perform predictions on several types of data. Neural Networks (NNs), and in particular Convolutional Neural Networks (CNNs), have emerged as a promising solution for many real-life problems such as facial recognition [110], image and video analysis for self-driving cars [29] and even for playing games (most readers probably know of *AlphaGo* [137] which in 2016 beat one of the top Go players). CNNs have also found applications within areas of medicine. [68], for example, demonstrates that CNNs are as effective as experts at detecting skin cancers from images, and [66] investigated using CNNs to examine chest x-rays.

Many applications that use Machine Learning to infer something about a piece of data, does so on data of sensitive nature, such as in the two examples cited above. In such cases the ideal would be to allow the input data to remain private. Conversely, and since model training is by far the most expensive part of deploying a model in practice,[1] preserving model privacy may be desirable as well.

In order to break this apparent contradiction (performing computation on data that is ought to be kept secret) tools like *secure multiparty computation* (MPC) can be used. Using such tools, image classification can be performed so that it discloses neither the image to the model owner, nor the model to the input owner. In the client-server model this is achieved by letting the data owner and the model owner *secret-share* their input towards a set of servers, who then run the computation over these shares.

Towards Deploying Secure Inference.

Research in the area of secure evaluation of CNNs has been rich during the last couple of years [80, 100, 109, 112, 118, 130–132, 142]. The main goal of this prior research has been to reduce the performance gap between evaluating a CNN in the clear and doing it securely. Current state of the art solutions rely on for example Garbled Circuits [131] or MPC [109]. Both of these works manage to evaluate large ImageNet type models (tens of layers and 1000 classes) with reasonable efficiency. Moreover, the CrypTFlow framework [109] also support secure inference with malicious security albeit by relying on a secure hardware assumption. Several solutions have also been developed by researchers closer to the industry side, such as TF-Encrypted [52] or CrypTen [70], which suggests that secure prediction has value beyond the academic point of view—an important factor for accelerating wide adoption of these techniques.

[1]The network by Yang et al. [144] costs between \$61 000 and \$250 000 to train according to `https://syncedreview.com/2019/06/27/the-staggering-cost-of-training-sota-ai-models/`.

These advances mean that secure evaluation of large models (tens of layers, millions of parameters) can now be performed in the order of seconds which, while too slow for real-time classification, is acceptable for many privacy critical tasks like the ones described in [66, 68]. Medical tests typically take hours if not days or weeks, so running an additional test that takes a few seconds or minutes will not matter.

Given the intense interest in secure inference in recent years, one might ask: *what obstacles are preventing the use of secure inference in practice?*. Most of the focus on the research literature has been on improving performance, but this is far from being the only challenge in this direction.

Challenges, the ML perspective. Setting aside the privacy requirement for a minute, deploying a Machine Learning solution in practice is already a long and strenuous process. Data has to be acquired and then processed; a model has to be designed, its parameters have to be tuned and finally the model needs to be trained. Moreover, the process is often repeated totally or in parts whenever new data is acquired or a better model design is found.

It is not surprising that a *significant* amount of effort is being spent on designing feature-rich and well documented frameworks for developing these models, for training them and for testing them. Frameworks such as TensorFlow [69], PyTorch [123] and MXNet [44] are all seeing significant use and are being actively developed by major companies (Google for TensorFlow, Facebook for PyTorch and Apache for MXNet).

Ideally, it should be possible to design a model using these widely used frameworks and then simply alter the way it is used at the very end when the model is used to perform predictions on user provided inputs. However, most existing solutions for secure inference throws a wrench into this process as they rely either on models that use specialized activation functions, such as CryptoNets [80] or require a specialized training process, such as XONN [131]. For these reasons, a part that is often of the *least* concern when developing a Machine Learning solution (using the final model for prediction) thus becomes an aspect that has to be considered at virtually all steps of the design process.

It is worth noting here that it is not enough for a framework to simply support conversion from a trained model into a representation that can be used. Such conversions typically involve steps like moving from a floating point representation to a fixed point one, or exchanging certain activation functions with “approximations”—all of which impact accuracy and thus the expressiveness of the model.

These issues were also identified by the authors of CrypTFlow [109] and a third of their work is spent detailing a customized application that converts TensorFlow code into a representation which can be run securely by their framework, without substantially compromising on accuracy.

Our work takes a different approach to this problem. Namely, we investigate

whether, and to what extent, existing frameworks such as TensorFlow support training and designing models that can be securely evaluated.

Challenges, the MPC perspective. In spite of notable progress in general-purpose MPC, most of the existing MPC-based secure inference works rely on customized subprotocols that are highly optimized for particular activation functions. Moreover, and as remarked on above, these activation functions are often themselves novel in the sense that they rarely see use outside of secure inference.

Such a tight coupling between what can be evaluated and how its evaluated also implies that often only a single threat model is supported. In particular, if a specific threat model is desired, then often this directly determines what kind of network that can be run. For example, if one requires a dishonest majority solution, such as a 2-party protocol, then XONN is the current most efficient solution. But XONN only works on binarized networks and only works with Sign as the activation function.

At the other end, if one is fine with honest majority (typically 3 parties, as that is the most efficient setting) but require active security, then only CrypTFlow fits the bill; but CrypTFlow relies on secure hardware for active security.

Ideally, the MPC that is used should be "oblivious" to the network being evaluated, as that would allow a user to more freely choose which threat model is best suited for them *without* having to think about the structure of the network, or specialized hardware. This is the approach we take. That is, we investigate how applicable general purpose MPC frameworks such as MP-SPDZ [103] or SCALE-MAMBA [7] are to evaluate Convolutional Neural Networks.

Using general-purpose MPC frameworks is not only convenient when it comes to the threat model. Another important factor is that, given their flexibility, this type of protocols tend to receive more scrutiny from part of the community [91], they are much better understood from a practical point of view and they have more reference implementations.[2] These considerations are important for an area like MPC, which is today in a stage in which many applications are within the practical realm, but no standardization of implementation practices (like the ones found e.g. in symmetric-key cryptography) are set yet.

[2] Although MPC protocols, general-purpose or not, are accompanied with security proofs, this does not mean that their "level of security" is the same. Details that appear only at implementation time, like instantiations of random oracles, make it important to have an end-to-end understanding of the security. Having reference implementations that are as close to industry-grade as possible is important, and the panorama in this regard for general-purpose protocols is much more promising than for special-purpose MPC.

Our Contribution

Our work addresses the challenges identified in the previous section, and to this end it focuses on the following two questions that, while simple in nature, have so far not been treated in research on secure inference.

1. To what extent can "MPC friendly" models be obtained from existing frameworks such as TensorFlow or PyTorch, *without* requiring a customized conversion protocol? More precisely, is it possible to design a model using these standard frameworks, which can then be *efficiently* evaluated by a secure protocol, without at all tampering with the model?

2. To what extent does existing MPC frameworks support running models "out-of-the-box"? That is, can we securely evaluate Machine Learning models (of the kind described above) using general-purpose MPC frameworks?

These questions provide a minimal baseline that one should keep in mind before moving on to specialized protocols or resorting to specialized models that improve efficiency. In this work we explore these questions thoroughly, which results in the following contributions:

Quantization. We identify the quantization techniques used in both TensorFlow, PyTorch and MXNet and described in [99] as particularly well suited for MPC. This type of quantization results in models for which each output of each convolutional layer can be expressed as a single dot-product followed by a truncation.

MPC. We describe how to implement these types models in a black-box way; that is, without resorting to special properties of the underlying MPC.

Optimizations. In settings where dot-products can be securely computed with high efficiency, the main bottleneck is truncation (or bitwise right-shift). As optimizations, we therefore present an optimized truncation protocol for some threat models which further improves efficiency.

Experiments. Finally, we evaluate the efficiency of a large class of quantized models in a variety of different threat models. More precisely, we evaluate 16 different models of varying size, each in 16 different settings.

We elaborate on each contribution below.

First, the fact that we identify a widely used quantization scheme as being particularly well suited for secure inference, provides a very promising area of study for both researchers and practitioners. For researchers, it provides a fixed target for secure protocol design. For practitioners, it shows that one does not have to abandon widely used Machine Learning frameworks in order to design models for practical use that can evaluated securely.

Secondly, showing that these techniques are compatible with the arithmetic black-box model (that is, only secure additions and multiplications are required) allows us to lower the requirements that an MPC protocol should satisfy in order to be suitable for secure inference. This in effect allows us to extend the amount of protocols supported. However, this does not mean that these protocols cannot be optimized, which in fact takes us to our third contribution: We present optimized primitives for the case of truncation over the ring $\mathbb{Z}_{2^k}$, which is not as well studied as the correponding problem over fields and constitutes the main bottleneck when securely evaluating our quantized neural networks.

Finally, to illustrate the advantages of our approach with respect to constructing ad-hoc MPC protocols that are only suitable to certain type of models, we perform a large amount of experiments with a wider range of models and different MPC protocols. More precisely, we securely evaluate 16 models that are part of the ImageNet family[3] using MPC protocols that vary with respect to several dimensions: Corruption threshold (honest vs. dishonest majority), corruption model (passive vs. active security) algebraic structure (integers modulo 2^k or modulo prime p) and whether or not truncation is exact or probabilistic.

Our experiments let us conclude several things:

1. Corruption threshold has a very large impact on efficiency for general-purpose MPC. Indeed, a dishonest majority evaluation takes orders of magnitude longer than honest majority.

2. On the other hand, corruption model has a comparatively smaller impact on efficiency. For example, an actively secure evaluation with an honest majority protocol over a prime field is only about 4 times slower than if the evaluation was done with the corresponding passive protocol.

3. For passive protocols we find that modulo a power of 2 is between 4 and 10 as efficient as the corresponding protocol modulo a prime power. This result further supports all the recent work that has gone into designing fast protocols that work over a ring [51, 62] (as this ring is typically taken to be integers modulo a power of 2).

4. Finally, by running our experiments for both exact truncation (meaning we evaluate the model *exactly* as would be done in the clear) and probabilistic truncation (meaning the evaluation may suffer some unknown loss in accuracy) we can quantify the exact gain in efficiency by relying on specialized protocols.

[3]The choice of evaluating these models is only because of convenience, as they can be found as part of the Tensorflow Lite model repository. As discussed in Section 5.7, *any* model trained with Tensorflow can be evaluated using our framework without the need of introducing extra tools.

Related Work

The following review is focused the different types of quantization (if any) that prior works has used; additionally, we look at frameworks for secure inference that prior works have developed. A broader review can be found in Appendix 5.7.

Quantization in prior work

Whether implicitly or explicitly, most prior work already uses some form of quantization. For instance, replacing floating-point by fixed-point numbers can be seen as a form of quantization. More often than not, however, this conversion is done in a very naive manner where the primary goal has been to fit the model parameters to the secure framework without further consideration about any potential impact it might have on the model's accuracy. Relatively little work has made explicit use of quantization in the context of securely evaluating Machine Learning models. One example is the recent work by Bourse et al. [31], where the authors use a quantization technique that is similar to the one described by Courbariaux and Bengio [47]. Sanyal et al. [133] use the same techniques. Nevertheless, their work lies in the FHE domain, which differs from multiparty computation. For instance, the fact that the weights are kept in the clear by the model owner changes the way the computation is performed, and allows them to use only additions and subtractions. XONN [131], which is based on Garbled Circuits, uses a quantization scheme which converts weights into bits [96]. For this to work, the authors need to increase the number of neurons of the network and a large part of their work is dedicated to describing how this scaling can be performed. CrypTFlow [109] employ what can be seen as a custom fixed-point-to-floating-point conversion protocol (called Athos) that automatically converts the floating point weights of a Tensorflow model into a fixed points representation, where the parameters are chosen so as to not compromise on the models original accuracy.

Frameworks for secure evaluation

Several previous works provide what can be viewed as a more complete framework for secure evaluation. The first of these is MiniONN [112] which provides techniques for converting existing models into models that can be evaluated securely. The authors demonstrate this framework by converting and running several models for interesting problem domains, such as Language modeling, as well as more standard problems such as hand writing recognition (MNIST) and image recognition (CIFAR10). CrypTFlow [109] also provides more complete framework. As already mentioned above, the first step in their framework is a protocol for converting an Tensorflow trained model into a model that can later be evaluated securely using a protocol based on SecureNN [142]

Outline of the Document

In Section 4.2 we give a brief introduction to Neural Networks after which we describe the quantization scheme we will be using. In Section 4.3 we provide a self-contained description of our protocol for secure inference, describing the basic building blocks. We discuss implementation details and present benchmarks in Section 5.7, and conclude in Section 4.5.

4.2 Deep Learning and Quantization

Deep learning models are at the core of many real-world tasks like computer vision, natural language processing and speech recognition. However, in spite of their high accuracy for many such tasks, their usage on devices like mobile phones, which have tight resource constraints, becomes restricted by the large amount of storage required to store the model and the high amount of energy consumption when carrying out the computations that are typically done over floating-point numbers. To this end, researchers in the machine learning community have developed techniques that allow weights to be represented by low-width integers instead of the usual 32-bit floating-point numbers, and quantization is recognized to be the most effective such technique when the storage/accuracy ratio is taken into account.

Quantization allows the representation of the weights and activations to be as low as 8 bits, or even 1 bit in some cases [47, 128].This is a long-standing research area, with initial works already dating back to the 1990s [14, 72, 114, 139], and this extensive research body have enabled modern quantized neural networks to have essentially the same accuracy as their full-precision counterparts [48, 85, 90, 122, 147].

Notation

For a value $\underline{x} \in \mathbb{R}^{N_1 \times N_2 \times N_3}$ we use $\underline{x}[i, j, c] \in \mathbb{R}$ to denote taking i'th value across the first dimension, the j'th value across the second dimension and the c'th value across the last dimension. In a similar way, we might write $\underline{x}[\cdot, \cdot, c] \in \mathbb{R}^{N_1 \times N_2}$ to denote the matrix obtained by fixing a specific value for the last dimension. A real value interval is denoted by $[a, b]$ and a discrete interval by $[a, b]_{\mathbb{Z}}$. We define *clamping* of a value $x \in \mathbb{R}$ to the interval $[a, b]$, denoted by $\mathsf{Clamp}_{a,b}(x)$, by setting $x \leftarrow a$ if $x < a$, $x \leftarrow b$ if $x > b$ and otherwise $x \leftarrow x$. (Clamping to a discrete interval is similarly defined.) We denote by $\mathbb{N}_\ell$ the set $\{1, \ldots, \ell\}$.

Deep Learning

A Convolutional Neural Network (CNN) is an ordered series of non-linear functions $(f_1, \ldots, f_n)$ where $f_i : D_{i-1} \mapsto D_i$ is called a *layer*, and where each

$D_i \in \mathbb{R}^{N_1 \times \cdots \times N_{m_n}}$ is a space of tensors. In practice, and for the networks we consider (ImageNet networks) $D_0 \in \mathbb{R}^{128 \times 128 \times 3}$ indicating that inputs are 128 by 128 pixel RGB images, and $D_n \in \mathbb{R}^{1000}$ indicates that there is 1000 output classes.

We are concerned mainly with the case where f_i is a convolution, followed by a Rectified Linear Unit (ReLU). More precisely, f_i can be expressed as $f_i(x) = \max(xW + b, 0)$ where W and b are tensors (*weights* and *bias*, respectively), and where max is applied entrywise.

Downsampling, i.e., making the height and width of the output of a layer smaller, can be achieved by applying pooling operations. Average pooling, for example, goes over windows of some size $w \times h$ in each channel of the input and outputs the average; that is the output $\underline{y}[i, j, c]$ will be the average of a $w \times h$ window centered around $\underline{x}[i, j, c]$, where $\underline{x}$ is the input tensor.

Finally, *batch normalization* [98] is often employed to speed up training. The idea is to normalize the inputs to each activation: instead of computing $g(x)$ for input x and activation function g, we instead compute $g(y)$ where

$$y = \gamma \left(\frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \beta, \tag{4.1}$$

where γ, β are parameters learned during training, and μ_B, σ_B^2 is the mean and variance, respectively, of a batch B of which x is a member. Consider an input $y = xW + b$ to g. During inference, we can "fold" the batch normalization parameters into the weights, which is done by using W' and b' defined as

$$W' = \frac{\gamma W}{\sigma}, \qquad b' = \gamma \left(\frac{b - \mu}{\sigma} \right) + \beta.$$

It is straight forward to verify that using $y' = xW' + b'$ yields the expression in Eq. (4.1).

Quantization of [99]

The goal of this section is to provide an overview of the quantization technique of Jacob et al. [99] (see also [108]) that we will be relying on to get efficient secure inference. While this particular quantization scheme might not be state of the art, or even the best for all choices of secure inference (e.g., XONN [131] relies on a different scheme to get efficient inference) we choose this particular scheme for the following reasons: It is implemented in Tensorflow (more precisely, TFLite [86]) and as such we get a user friendly, widely available and well documented way of training models that can be securely evaluated. The fact that Tensorflow can be used to directly train models for our framework is very handy indeed as it removes the need to develop custom tooling that has little to do with the secure framework itself. Moreover, Tensorflow provides several

pre-trained ImageNet models which provides a very good point of reference for not only our benchmarks, but for future works that wish to compare against us. Indeed, few if any previous work on secure inference provide pretrained models which makes an accuracy oriented comparison very hard.[4] Our focus here is a particular quantization scheme; for a broader survey, we refer the reader to Guo [89].

We note that this scheme is beneficial for MPC since it simplifies the activations and the arithmetic needed to evaluate a CNN. However, the original goal of Jacob et al. was to reduce the size of the models, rather than simplifying the arithmetic or the activations. Unfortunately, we do not get the benefits in the size reduction since, even if the network can be stored using 8-bit integers, arithmetic must be done modulo 2^{32} and even 2^{64} in some cases.

Quantization and De-Quantization

The scheme comes in two variants, one for 8-bit integers and another one for 16-bit integers. In this work we focus in the former, and we provide our description only in that setting.

Let $m \in \mathbb{R}$ and $z \in [0, 2^8)_{\mathbb{Z}}$ and consider the function $\mathsf{Quant}^{-1}_{m,z} : [0, 2^8)_{\mathbb{Z}} \to \mathbb{R}$ given by $\mathsf{Quant}^{-1}_{m,z}(x) = m \cdot (x - z)$. This function transforms the interval $[0, 2^8)_{\mathbb{Z}}$ injectively into the interval $I = [-m \cdot z, m \cdot (2^8 - 1 - z))$ and as such it admits and inverse $\mathsf{Quant}_{m,z}$ mapping elements in the image of $\mathsf{Quant}^{-1}_{m.z}$ into $[0, 2^8)_{\mathbb{Z}}$. We define the quantization of a number $\alpha \in I$ to be $\mathsf{Quant}_{m,z}(\alpha')$, where α' is closest number to α such that α' is in the image of $\mathsf{Quant}^{-1}_{m,z}$.

The constants m, z above are the parameters of the quantization, and are known as the *scale* and the *zero-point*, respectively. This quantization method will be applied on a per-tensor basis, i.e. each individual tensor $\boldsymbol{\alpha}$ has a single pair m, z associated to it. These parameters are determined at training time by recording the ranges on which the entries of a given tensor lie, and computing m, z such that the interval $[-m \cdot z, m \cdot (2^8 - 1 - z))$ is large enough to hold these values. See Figure 4.1 for a visualization of this quantization method, and see Jacob et al. [99] for details.

Dot Products

Computing dot products is a core arithmetic operation in any CNN. In this section we discuss how to do this with the quantization method described above.

Let $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_N)$ and $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_N)$ be two vectors of numbers with quantization parameters (m_1, z_1) and (m_2, z_2), respectively. Let $\gamma = \sum_{i=1}^{N} \alpha_i \cdot \beta_i$, and suppose that γ is part of a tensor whose quantization parameters are (m_3, z_3). Let $c = \mathsf{Quant}_{m_3,z_3}(\gamma)$, $a_i = \mathsf{Quant}_{m_1,z_1}(\alpha_i)$ and $b_i = \mathsf{Quant}_{m_2,z_2}(\beta_i)$.

[4] This is especially the case if it is not clear exactly how the model was trained and which training and test data was used.

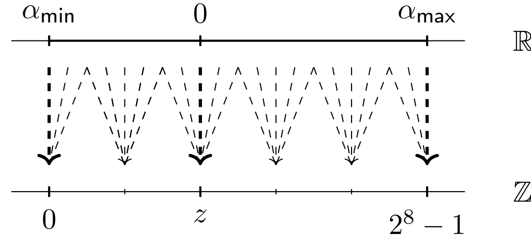


Figure 4.1: Visualization of the Quantization Scheme by Jacob et al. [99]. The continuous interval on top is mapped to the discrete interval below, and multiple numbers may map to the same integer due to the rounding.

It turns out we can compute c from all the a_i, b_i by using integer-only arithmetic and fixed-point multiplication, as shown in the following.

Since $\gamma \approx m_3 \cdot (c - z_3)$, $\alpha_i \approx m_1(a_i - z_1)$ and $\beta_i \approx m_2(b_i - z_2)$, it holds that

$$m_3 \cdot (c - z_3) \approx \gamma = \sum_{i=1}^{N} \alpha_i \cdot \beta_i \approx \sum_{i=1}^{N} m_1 \cdot (a_i - z_1) \cdot m_2 \cdot (b_i - z_3).$$

Hence, we can approximate c as

$$c = z_3 + \frac{m_1 \cdot m_2}{m_3} \cdot \sum_{i=1}^{N} (a_i - z_1) \cdot (b_i - z_2) \tag{4.2}$$

The summation $s = \sum_{i=1}^{N}(a_i - z_1) \cdot (b_i - z_2)$ involves integer-only arithmetic and it is guaranteed to fit in $16 + \log N$ bits, since each summand, being the product of two 8-bit integers, fits in 16 bits. However, since $m = (m_1 m_2)/m_3$ is a real, the product $m \cdot s$ cannot be done with integer-only arithmetic. This product is handled in TFLite by essentially transforming m into a fixed-point number and then performing fixed-point multiplication, rounding to the nearest integer. More precisely, m is first normalized as $m = 2^{-n} m''$ where $m'' \in [0.5, 1)$,[5] and then m'' is approximated as $m'' \approx 2^{-31} m'$, where m' is a 32-bit integer. This is highly accurate since $m'' \geq 1/2$, so there are at least 30 bits of relative accuracy.

Thus, given the above, the multiplication $m \cdot s$ is done by computing the integer product $m \cdot s$, which fits in 64 bits since both m and s use at most 32 bits (if $N \leq 2^{16}$), and then multiplying by 2^{-n-31} followed by a rounding-to-nearest operation. Finally, addition with z_3 is done as simple integer addition.

If the quantization parameters for γ were computed correctly, it should be the case, by construction, that the result c lies in the correct interval $[0, 2^8)_{\mathbb{Z}}$.

[5] Jacob et al. [99] find that in practice $m \in [0, 1)$, which is the reason why such normalization is possible. We also confirm this observation in our experiments, although it is not hard to extend this to the general case (in fact, TFLite already supports it).

However, due to the different rounding errors that can occur above, this may not be the case. Thus, the result obtained with the previous steps is clamped into the interval $[0, 2^8)_{\mathbb{Z}}$.

Addition of bias

In the context of CNNs the dot products above will come from two-dimensional convolutions. However, these operations not only involve dot products but also the addition of a single number, the bias. In order to handle this in a smooth manner with respect to the dot product above, the scale for the bias is set as $m_1 m_2 / m_3$ and the zero-point it set to 0. This allows the quantized bias to be placed inside the summation s, involving no further changes to our description above.

Other layers

Other layers like ReLU, ReLU6 or max pooling, which involve only comparisons, can be implemented with relative ease directly on the quantized values, assuming these share the same quantization parameters. This is because if $\alpha = m(a - z)$ and $\beta = m(b - z)$, then $\alpha \leq \beta$ if and only if $a \leq b$, so the comparisons can be performed directly on the quantized values.

In fact, activations like ReLU6 (which is used extensively in the models we consider in this work) can be entirely fused into the dot product that precedes it, as shown in Section 2.4 of [99]. Since ReLU6 is essentially a clamping operation, it is possible, by carefully picking the quantization parameters, to make the clamping of the product to the interval $[0, 2^8)_{\mathbb{Z}}$ *also* take care of the ReLU6 operation. In short, if the zero-point is 0 and the scale is 6/255, then we are guaranteed that $m(q - z) \in [0, 6]$ for any $q \in \{0, \ldots, 2^8 - 1\}$.

On the other hand, mathematical functions like sigmoid must be handled differently. We will not be concerned with this type of functions in this document since it is the case in practice that ReLU and ReLU6 (or similar activation functions) are typically enough.[6]

4.3 Quantized CNNs in MPC

In the previous section we discussed how quantization of neural networks works, or, more specifically, we discussed the quantization scheme by Jacob et al. [99]. Now, we turn to the discussion about how to implement these operations using MPC. However, before diving into the details of the protocols we use in this work, we describe the setting we consider for the secure evaluation of CNNs.

[6] See [99] for a discussion on quantization of mathematical functions.

System and Threat Model

Like most previous work on secure inference using MPC, we consider a setting where both the model owner and client outsource their model, respectively input to a set of servers that perform that actual secure inference.

We consider a setting of either two or three servers P_1, P_2 and P_3 depending on the setting (honest or dishonest majority) among which one is allowed to be corrupted. The model and input owner each secret-share their inputs to the servers at the beginning of the protocol execution. This preserves the privacy of this sensitive information under certain assumptions on the adversarial corruption. Then, the servers execute a secure multiparty computation protocol to evaluate the quantized model on the given input, obtaining shares of the output, which can then be sent to the party that is supposed to get the classification result.

As we already mentioned previously, our techniques have the crucial feature that virtually *any* secret-sharing-based MPC protocol can be used as the underlying computation engine. More precisely, let R be either $\mathbb{Z}_{2^k}$ or $\mathbb{F}_p$, we only assume a secret-sharing scheme $[\![\cdot]\!]$ over R for two or three parties (depending on the setting) withstanding one corruption, allowing local additions $[\![x+y]\!] \leftarrow [\![x]\!] + [\![y]\!]$, together with a protocol for secure multiplication $[\![x \cdot y]\!] \leftarrow [\![x]\!] \cdot [\![y]\!]$.[7]

The general-purpose MPC protocols we use in this work can be categorized according to three different dimensions: corruption threshold, type of corruptions and underlying algebraic structure. For the first dimension we distinguish between two cases: honest vs dishonest majority. In the former, the adversary is allowed to corrupt strictly less than half of the parties. We instantiate this case with 3 parties and 1 corruption, as that leads to the most efficient protocols. In the latter case, the adversary can corrupt any number of parties provided at least one party remains honest. We instantiate this setting for 2 parties. While honest majority protocols impose a stronger security assumption than dishonest majority, they tend to be simpler in their design and thus more efficient. We further distinguish between passive and active corruptions, where the former means the adversary follows the protocol and the latter allows the adversary to deviate. Not surprising, actively secure protocols impose an overhead over passively secure ones. Finally, the algebraic structure on which the computation takes place also plays an important role in terms of efficiency and protocol design, with protocols over $\mathbb{F}_p$ being easier to design and possibly implement, but protocols over $\mathbb{Z}_{2^k}$ providing some efficiency improvements in terms of basic arithmetic and bit-operations [61]

We consider a total of 8 MPC protocols to support the secure evaluation of the quantized CNNs, corresponding to all the possible combinations of the

[7] Protocols with these features are typically referred to as general-purpose MPC protocols, and any construction that only makes use of these properties is said to be in the arithmetic black-box model.

Threshold		$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$
$t < n$	Passive	OTSemi2k	OTSemiPrime
	Active	SPDZ2k	LowGear
$t < n/2$	Passive	Replicated2k	ReplicatedPrime
	Active	PsReplicated2k	PsReplicatedPrime

Table 4.1: MPC protocols we use, classified depending on their security level (passive vs. active) and their arithmetic properties (modulo 2^k vs. modulo a prime). Names are from MP-SPDZ.

three dimensions mentioned above (active/passive, honest/dishonest majority and computation modolu a prime or a power-of-two). Table 4.1 contains an overview of which protocol is used in which security model. We provide more details on each protocol in Section 5.7 in the appendix.

Building Blocks

For many applications, the multiplication protocol assumed for $[\![\cdot]\!]$ is not enough. In practice, many useful functionalities cannot be nicely expressed in terms of additions and multiplications and therefore, more often than not, researchers end up developing custom protocols for specific applications. As we argued in Section 5.1, this also includes the case of secure evaluation of Neural Networks.

In our case, thanks to the quantization scheme by Jacob et al. [99] most of the operations in the evaluation of a quantized Neural Network become additions and multiplications, which are already supported by the MPC protocols we consider here. Furthermore, the multiplications have a very special structure: they are part of a dot product operation, which can be computed more efficiently for the particular case of passive security with an honest majority. However, the evaluation still requires non-arithmetic operations like truncations and comparisons, which are more expensive and require specialized subprotocols for their computation. These, fortunately, can be also implemented in the arithmetic black-box model, that is, making use only of additions and multiplications, which preserves our flexibility when it comes to the underlying MPC protocol. In what follows we describe the primitives we require in order to integrate the quantized models from Section 4.2 into our secure engine.

Secure comparison

An important primitive involves comparing two secret-shared values, in order to take certain action depending on which of the two is larger. However, since revealing which of the two inputs is larger leaks information about the inputs themselves (which is not allowed in many applications), a secure comparison protocol outputs the bit indicating the result of the comparison in secret-shared

form. More precisely, a secure comparison subprotocol allows the parties to compute $[\![b]\!] \leftarrow [\![x]\!] \overset{?}{<} [\![y]\!]$, that is, $b = 1$ if $x < y$, and $b = 0$ otherwise.

Just like the case with truncation, this problem is well motivated and has received enough attention by the community, with many existing proposals providing different trade-offs. Given this, we may assume the existence of a secure comparison subprotocol, which can be instantiated for example using the constructions from [38] for the field case, and [62] for the ring case. For the special case of replicated secret sharing over $\mathbb{Z}_{2^k}$, Mohassel and Rindal [117] have proposed a more efficient approach. Comparison is equivalent to extracting the most significant bit of the difference between the two operands. This bit can be computed from the carry bit of adding the three shares, which in turn is possible to achieve by a binary circuit on the local bit decomposition of shares. While this binary circuit has linear complexity in the bit length, it only takes one bit to compute an AND gate in this setting.

Truncation by a public value

As we have already discussed in the introduction, most existing works in the area of secure inference make use of fixed-point arithmetic, in which a rational number α is approximated by the closest integer to $\alpha \cdot 2^t$, where t is some *fixed* parameter. To keep the right representation after multiplying two fixed-point numbers, the result must be truncated by t bits, which is a non-linear operation and generates several complexities when done in MPC.

Many solutions have been developed throughout the years for computation over both the field $\mathbb{F}_p$ and the ring $\mathbb{Z}_{2^k}$. We refer the reader to [38] and [62] for details on these. For the purpose of our work, we assume the existence of a subprotocol that computes $[\![y]\!] \leftarrow [\![x]\!]$, where $y = \lfloor \frac{x}{2^m} \rfloor$, where m is some fixed, *public* parameter.

It is also useful to consider the concept of *probabilistic truncation*. In this case, instead of obtaining $[\![y]\!]$ from $[\![x]\!]$, where $y = \lfloor \frac{x}{2^m} \rfloor$, a protocol for probabilist truncation computes $[\![z]\!]$ where $z = \lfloor \frac{x}{2^m} \rfloor + u$ and u is some small error. In practice, $u \in \{0, 1\}$, and u is "biased towards $\lfloor \frac{x}{2^m} \rceil$", meaning that u equals 1 with probability the decimal part of $\frac{x}{2^m}$, which equals $(x \bmod 2^m)/2^m$. As an example, if $x = 7$ and $m = 2$, a protocol for probabilistic truncation would produce either $\lfloor \frac{7}{4} \rfloor = 1$ or $\lfloor \frac{7}{4} \rfloor + 1 = 2$, where the latter happens with probability .75.

Since neural networks tend to be quite resilient to small changes in the activations, which as we will see is the ultimate effect of having probabilistic truncation instead of deterministic (i.e. exact), this approach should not affect the accuracy of the models substantially, although we do not verify this experimentally. Furthermore, probabilistic truncation protocols tend to perform much better than deterministic ones, as we show experimentally in Section 5.7. This is because these protocols avoid the usage of expensive binary adders and other similar binary circuits that appear in the deterministic case. See [38] for

some details. In what follows we introduce some novel protocols for the task of probabilistic truncation over a ring $\mathbb{Z}_{2^k}$.

Probabilistic truncation over the ring $\mathbb{Z}_{2^k}$. Truncation over the ring $\mathbb{Z}_{2^k}$ is considerably more difficult as truncation over $\mathbb{F}_p$, as the latter relies on the fact that division by 2^m can be done simply by locally multiplying by the inverse of 2^m, which is not possible over $\mathbb{Z}_{2^k}$. The authors in [62] and [117] propose alternative methods to deal with this issue, but unfortunately their methods require either non-constant number of rounds, or require a large gap between the shares and the secret, which hurts performance.

Instead, we propose a novel method to perform secure truncation over $\mathbb{Z}_{2^k}$, where the shares only need to be one bit larger than the secrets and the number of rounds is constant. The result may have an error of at most 1, but this error is biased towards the nearest integer to $x/2^m$, where x is the value being truncated. In our protocol we assume a method to produce random shared bits, that is, $[\![b]\!]$ where $b \in \{0, 1\}$ is uniformly random and unknown to the adversary. This can be done in the dishonest majority setting as proposed in [62], or more generally, we can let each party P_i propose a bit $[\![b_i]\!]$ (which can be checked to be a bit indeed by verifying that $[\![b_i]\!] \cdot (1 - [\![b_i]\!])$ is 0) and then the parties XOR these bits together to get one single random bit.

Protocol 1 $\mathsf{TruncPr}_{\mathbb{Z}_{2^k}}([\![x]\!], m)$

Pre: x with $\mathsf{MSB}(x) = 0$.

Post: $[\![x/2^m]\!]$ rounded according to text.

Proceed as follows:

1. Generate k random shared bits $[\![r_i]\!]$ and compute $[\![r]\!] \leftarrow \sum_i [\![r_i]\!] \cdot 2^i$.
2. Open $c \leftarrow [\![x]\!] + [\![r]\!]$ and compute $c' \leftarrow (c/2^m) \bmod 2^{k-m-1}$.
3. Compute $[\![b]\!] \leftarrow [\![r_{k-1}]\!] \oplus (c/2^{k-1})$.
4. Output $c - \sum_{i=m}^{k-2} [\![r_i]\!] \cdot 2^{i-m} + [\![b]\!] \cdot 2^{k-m-1}$.

An improvement for the ring case with three parties, honest majority and passive security. By further restricting the setting, more optimizations can be done. We consider replicated secret sharing over $\mathbb{Z}_{2^k}$ with three parties and passive security. Our truncation protocol emulates the black-box probabilistic truncation in the setting of semi-honest computation over a power of two with an honest majority. Informally, it changes from a symmetric three-party protocol to a two-party protocol where the third party generates correlated

randomness used by the the other parties. This allows to generate random values of any bit length at once without the need to generate such random values bit-wise. The latter is the main cost in black-box probabilistic truncation because the communication is independent of the number of bits otherwise.

Protocol 2 $\mathsf{TruncPrSp}_{\mathbb{Z}_{2^k}}([\![x]\!], m)$

P_3 proceeds as follows:

1. Sample random bits $\{r_i\}$ for $i \in [0, k-1]$.
2. Generate 2-out-of-2 sharings of $r = \sum_i r_i \cdot 2^i$, r_{k-1}, and $\sum_{i=m}^{k-2} r_i \cdot 2^{i-m}$, and send one share to P_1 and P_2 each.
3. Generate random $y_1, y_3 \in \mathbb{Z}_{2^k}$ and send y_1 to P_1 and y_3 to P_2.
4. Output (y_3, y_1).

P_1 and P_2 proceed as follows:

1. Convert $[\![x]\!]$ to a 2-out-of-2 sharing by P_1 computing $x_1 + x_2$ and P_2 proceeding with x_3.
2. Execute $\mathsf{TruncPr}_{\mathbb{Z}_{2^k}}$ as two-party computation using the random values received from P_3.
3. P_i: Let y'_i denote the share output by $\mathsf{TruncPr}_{\mathbb{Z}_{2^k}}$ and $\hat{y}_i$ the share received from P_3 (y_1 or y_3). Send $y'_i - \hat{y}_i$ to P_{2-i}. Denote the received value by $\tilde{y}_i$.
4. P_1 outputs $(y_1, y'_1 - \hat{y}_1 + \tilde{y}_1)$, and P_2 outputs $(y'_2 - \hat{y}_2 + \tilde{y}_2, y_3)$.

For correctness, we have to establish that the parties output a correct replicated secret sharing of the result. To establish the correct replicated secret sharing, consider

$$\begin{aligned} y'_1 - \hat{y}_1 + \tilde{y}_1 &= y'_1 - \hat{y}_1 + y'_2 - \hat{y}_2 \\ &= \tilde{y}_2 + y'_2 - \hat{y}_2. \end{aligned}$$

Furthermore,

$$\begin{aligned} y_1 + y_3 + y'_1 - \hat{y}_1 + \tilde{y}_1 &= y_1 + y_3 + y'_1 - \hat{y}_1 + y'_2 - \hat{y}_2 \\ &= y_1 + y_3 + y'_1 - y_1 + y'_2 - y_2 \\ &= y'_1 + y'_2, \end{aligned}$$

which equals the result of $\mathsf{TruncPr}_{\mathbb{Z}_{2^k}}$ by definition.

Since we only aim for semi-honest security with honest majority, we have to show that each party does not learn any information about x if all parties follow the protocol. This is trivial for P_3 because they do not receive anything. For P_1 and P_2, the randomness received from P_3 is independent of x. Furthermore, the security of the two-party $\mathsf{TruncPr}_{\mathbb{Z}_{2^k}}$ execution follows by the black-box definition of it. Finally, $\tilde{y}_i$ does not reveal information because $\hat{y}_{2-i}$ is uniformly random and unknown to P_i.

Truncation by a Secret Value

The truncation protocols we have considered so far assume that the amount of bits to be truncated, m, is public. This is a natural setting and appears for instance in fixed-point multiplication, where m is equal to the amount of bits assigned for the decimal part. However, as we already argued in Section 4.2, the quantization scheme we use here differs from traditional fixed-point arithmetic in that the parameters for the discretization are adaptively chosen for each particular layer of the network. As a side effect, these parameters become information of the model, and therefore they must not be revealed in the computation. As a result, truncation by secret amounts become necessary.

In this section we present our protocol for truncation by a secret amount. It takes as input a secret $[\![x]\!]$ and a shift m represented by $[\![2^{M-m}]\!]$ where M is some public upper bound on m,[8] and outputs $[\![y]\!]$ where $y = \lfloor \frac{x}{2^m} \rfloor$.

Protocol 3 $\mathsf{truncp}_R([\![x]\!], [\![m]\!])$

The parties proceed as follows.

1. Compute $[\![2^{M-m} \cdot x]\!] = [\![2^{M-m}]\!] \cdot [\![x]\!]$.
2. Return $[\![y]\!] \leftarrow \mathsf{Trunc}_R([\![2^{M-m} \cdot x]\!], M)$.

Security. We informally argue that the shift m used in the protocol remains hidden. To this end, simply notice that m is provided as input to the MPC protocol in secret-shared form $[\![m]\!]$. Then, the multiplication $[\![2^{M-m} \cdot x]\!] = [\![2^{M-m}]\!] \cdot [\![x]\!]$ does not leak anything since we assume the underlying multiplication protocol is secure. Finally, since we assume the protocol for *public* truncation Trunc_R is secure, the call $[\![y]\!] \leftarrow \mathsf{Trunc}_R([\![2^{M-m} \cdot x]\!], M)$ produces correct shares without leaking anything, which implies that $y = \lfloor \frac{2^{M-m}x}{2^M} \rfloor = \lfloor \frac{x}{2^m} \rfloor$.

[8] We may alternatively assume that m itself is shared. The conversion $[\![m]\!] \rightarrow [\![2^{M-m}]\!]$ can be achieved then by first bit-decomposing $M - m$ as $\sum_i 2^i \cdot b_i$, computing shares of each b_i and then outputting $[\![2^{M-m}]\!] = \prod_i (1 + [\![b_i]\!] \cdot (2^{2^i} - 1))$. However, since in our setting m is known by the client who has the model, it is simpler to assume that the client distributes $[\![2^{M-m}]\!]$ to begin with.

The only requirement for this protocol to work is that $(M - m) + \log_2(|x|)$ must be smaller than bit-length of the modulus of the secret sharing scheme since, in this case, it can be seen that $2^{M-m} \cdot x$ does not overflow.

Putting it all Together

Using the building blocks that we just described, together with the quantization scheme from Section 4.2, we can securely evaluate quantized neural networks in an easy way. As we discussed in Section 4.2, evaluating a quantized CNN consists mostly of computing the expression in Eq. (4.2), followed by a clamping procedure. We describe these computations in this section, along with the other necessary pieces for the evaluation of a quantized CNN.

Recall from Section 4.2 that each weight tensor $\underline{a}$ in a quantized CNN has a scale $m \in \mathbb{R}$ and a zero-point $z \in \mathbb{Z}_{2^8}$ associated to it, such that $\alpha \approx m \cdot (a - z)$ is the actual floating-point numbers corresponding to each 8-bit integer a in the tensor. Also, biases are quantized in a similar manner but with a 32-bit integer instead, a zero point equal to 0, and a scale that depends on the inputs and output to the layer it belongs to, as explained in Section 4.2. We assume that the model owner, who knows all this information, distributes shares to the servers using the scheme described above of the quantized weights and biases of each layer in the network.[9] Also, the zero points associated to each tensor are shared towards the parties.

The scales of the model, on the other hand, are handled in a slightly different way. Each dot product in the quantized network requires a fixed-point multiplication by a factor $m = (m_1 \cdot m_2)/m_3$, borrowing the notation from Section 4.2. Recall that this product was handled by writing $m = 2^{-n-31} \cdot m'$, where m' is a 32-bit integer.

Now, to compute securely the expression in Eq. (4.2), recall that the parties have shares of the zero points z_1, z_2, z_3, the quantized inputs a_i, b_i for $i = 1, \ldots, N$, the integer scale m' and the power $2^{L-\ell}$, where $\ell = n + 31$ with $2^{-n-31} \cdot m' \approx m = (m_1 \cdot m_2)/m_3$, and L is an upper bound on ℓ.[10] To compute Eq. (4.2), the parties begin by computing the dot product $[\![s]\!] = \sum_{i=1}^{N}([\![a_i]\!] - [\![z_1]\!]) \cdot ([\![b_i]\!] - [\![z_2]\!])$. Then, an additional secure multiplication is used in order to compute $[\![m \cdot s]\!] = [\![m]\!] \cdot [\![s]\!]$. Next, shares of $\lfloor 2^{-n-31} \cdot (m \cdot s) \rceil$ are computed from $[\![2^{L-\ell}]\!]$ and $[\![m \cdot s]\!]$ using Protocol `truncp` from Section 4.3, together with the observation that $\lfloor 2^{-m} \cdot x \rceil = \lfloor 2^{-m} \cdot x + 0.5 \rfloor = \lfloor 2^{-m} \cdot (x + 2^{m-1}) \rfloor$ for breaking a tie by rounding up.

[9] Notice that these values are only 8-bit long in the clear, but the shares are 64-bit long. The reason is that, although the values are small, the computation must be carried without overflow. Therefore we cannot use a modulus that is smaller than the maximum possible intermediate value.

[10] Since $n \leq 32$ it suffices to take $M = 63$. In this case, given that $m \geq 31$, it follows that $2^{M-m} \leq 2^{32}$. According to Section 4.3, this imposes the restriction that the modulus for the computation must be at least $32 + 64 = 96$. In practice n is smaller than 32 and this bound can be improved.

Finally, addition with $[\![z_3]\!]$ is local, and it is followed by the clamping the result $[\![x]\!]$ to the interval $[0, 2^8)$. This is done by comparing $[\![x]\!]$ to the limits (0 and 255) using a secure comparison protocol (see Section 4.3), followed by an oblivious selection: If $s \in \{0, 1\}$, it holds trivially that $a_s = s \cdot (a_1 - a_0) + a_0$ for arbitrary a_0, a_1.

Other layers. Average pooling involves computing $[\![y]\!]$ from $[\![x_1]\!], \ldots, [\![x_n]\!]$, where $y = \lfloor \frac{1}{n} \cdot \sum_{i=1}^{n} x_i \rceil$. This can be achieved using Goldschmidt's algorithm [83], a widely used iterative algorithm for division. For its usage in the context of secure multiparty computation, see for example Catrina and Saxena [39]. It uses basic arithmetic as well as truncation, both of which we have already discussed.

On the other hand, max pooling requires implementing the max function securely, which can be easily done by making use of a secure comparison protocol [38].

Finally, once shares of the output vector are obtained (raw output, before applying Softmax), several options can be considered. The parties could open the vector itself towards the input owner and/or data owner so that they compute the Softmax function and therefore learn the probabilities for each label. However, this would reveal all the prediction vector, which could be undesirable in some scenarios. Thus, we propose instead to securely compute the argmax of the output array, and return this index, which returns the most likely label since exponentiation is a monotone increasing function. Previous work, such as SecureML [118], replace the exponentiation in the Softmax function with ReLU operations, i.e. by computing $\mathsf{ReLU}(x)$ instead of e^x. More MPC friendly solutions exist, such as the spherical Softmax [64], which replaces e^x with x^2.

4.4 Implementation and Benchmarking

This section discusses our implementation and our performance results.

MobileNets

Our benchmarks are all performed by evaluating networks of the *MobileNets* type architecture [95]. A MobileNets network consists of 28 layers with 1000 output classes and are trained on the ImageNet data set [97]. Layers are alternating (with few exceptions at the start and end) *pointwise* convolutions and *depthwise* convolutions. A pointwise convolution is a regular convolution with a 1×1 filter, while a depthwise convolution can be viewed as a convolution where no summation across output channels occurs. The size of the network can be adjusted by two hyper parameters: a *width multiplier* α and a *resolution multiplier* ρ. α scales input and output channels, while ρ scales the dimensions

# sum-of-products	mod 2^k		mod p		mod p (active security)	
	Runtime (s)	Comm. (gb)	Runtime (s)	Comm. (gb)	Runtime (s)	Comm. (gb)
50 000	0.25	0.15	1.6	0.54	8.8	4.3
100 000	0.41	0.31	2.5	1.07	15.6	8.5
150 000	0.57	0.46	3.6	1.59	22.5	12.8
200 000	0.73	0.62	4.5	2.12	29.2	17.0
# of terms						
256	0.27	0.31	1.9	1.1	9.4	5.7
512	0.30	0.31	2.0	1.1	13.9	8.5
768	0.33	0.31	2.3	1.1	18.5	11.4
1024	0.36	0.31	2.4	1.1	22.9	14.3

Table 4.2: Top: running a variable number of sum-of-products of constant length $\ell = 512$. Bottom: Running $n = 100.000$ sum-of-products with variable length.

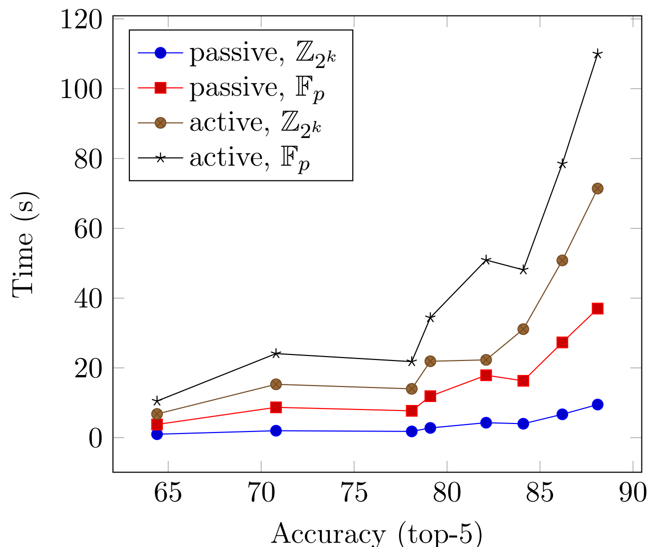


Figure 4.2: Evaluation times for honest majority protocols. x-axis generally correspond to evaluating larger models.

of the input image. Thus α reduces both the model size (as there will be fewer parameters with a smaller α) and number of operations, while ρ only scales the number of operations. In the following we denote a particular model as "`V1` α_S" where S is the height and width of the input image (thus dependent on ρ). We evaluate the pretrained models which are available on the Tensorflow repository,[11] and we use their accuracy values.

[11] See `https://www.tensorflow.org/lite/guide/hosted_models`

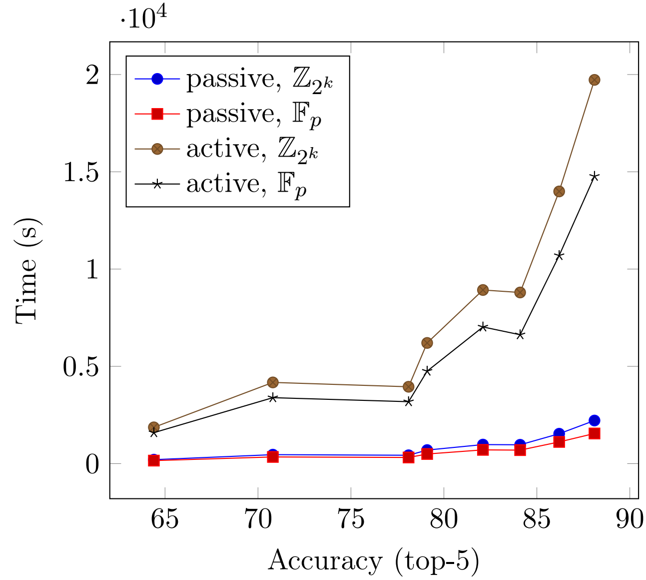


Figure 4.3: Evaluation times for dishonest majority protocols. x-axis generally correspond to evaluating larger models.

Quantizing arbitrary tensorflow models. We choose to evaluate the MobileNet models for two reasons: First, they can be considered realistic in the sense that they are expressive enough to solve a wide variety of image related classification tasks. Thus, the evaluation times we report in this section will correspond to running evaluations of similarly expressive models in practice. Second, the models are hosted in pre-trained form online which in principle makes our results reproducible (prior work, while sometimes describing the architecture of the models they evaluate, very rarely describe the training process).

However, our technique is by no means limited to running only MobileNet networks. A quantized model is obtained either by performing quantization aware training[12] or by post-quantizing an already trained model[13]. We stress that both these models are implemented entirely by TensorFlow, so no external conversion is needed.

Implementation

We implement secure inference in the MP-SPDZ framework, which allows us to get timings for all the protocols described in Section 5.7. These protocols run over either a prime p or a ring $\mathbb{Z}_{2^k}$. The prime is 128 bits while the k we use

[12]See `https://github.com/tensorflow/tensorflow/tree/r1.13/tensorflow/contrib/quantize`

[13]See `https://www.tensorflow.org/lite/performance/post_training_quantization`

Variant	Accuracy		Trunc.	Passive Security			
				Dishonest Maj.		Honest Maj.	
	Top-1	Top-5		$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$
V1 0.25_128	39.5%	64.4%	Prob.	139.5	129.1	0.2	3.3
			Exact	203.2	155.0	1.0	3.8
V1 0.25_160	42.8%	68.1%	Prob.	214.5	201.1	0.3	5.2
			Exact	317.7	241.2	1.4	6.1
V1 0.25_192	45.7%	70.8%	Prob.	305.1	288.5	0.4	7.3
			Exact	460.6	343.1	2.0	8.7
V1 0.25_224	48.2%	72.8%	Prob.	417.6	383.3	0.5	10.0
			Exact	614.1	460.8	2.9	11.8
V1 0.5_128	54.9%	78.1%	Prob.	305.4	267.0	0.4	6.5
			Exact	430.1	316.3	1.8	7.7
V1 0.5_160	57.2%	80.5%	Prob.	472.6	418.3	0.6	10.4
			Exact	672.1	496.5	2.9	12.5
V1 0.5_192	59.9%	82.1%	Prob.	676.1	593.1	0.9	15.2
			Exact	978.0	706.3	4.3	17.9
V1 0.5_224	61.2%	83.2%	Prob.	915.6	802.2	1.1	20.5
			Exact	1320.6	955.6	5.8	24.4
V1 0.75_128	55.9%	79.1%	Prob.	485.4	421.2	0.6	9.9
			Exact	697.3	494.5	2.8	11.9
V1 0.75_160	62.4%	83.7%	Prob.	775.7	662.2	1.1	15.9
			Exact	1075.8	779.5	4.6	19.0
V1 0.75_192	66.1%	86.2%	Prob.	1101.1	943.0	1.6	23.3
			Exact	1536.9	1114.8	6.7	27.3
V1 0.75_224	66.9%	86.9%	Prob.	1487.2	1276.8	2.2	31.4
			Exact	2135.5	1505.8	9.6	37.4
V1 1.0_128	63.3%	84.1%	Prob.	709.4	587.1	1.0	13.5
			Exact	968.5	694.1	4.0	16.3
V1 1.0_160	66.9%	86.7%	Prob.	1101.8	928.4	1.8	21.7
			Exact	1528.0	1084.0	6.5	25.9
V1 1.0_192	69.1%	88.1%	Prob.	1581.6	1323.9	2.6	31.5
			Exact	2214.8	1549.0	9.5	37.0
V1 1.0_224	70.0%	89.0%	Prob.	2147.3	1792.2	3.5	42.5
			Exact	2943.3	2101.4	13.1	50.4

Table 4.3: Running time, in seconds, of securely evaluating some of the networks in the MobileNets family with passive security, in a LAN network. The first number in variant is the width multiplier and the second is the resolution multiplier. Top-1 accuracy measures when the truth label is predicted correctly by the model whereas Top-5 measures when the truth label is among the first 5 outputs of the model. Prob. and Exact refer to probabilistic truncation and nearest rounding, respectively.

for the ring is 72 bits. As described in Section 4.3, these arise because we need some extra space in order for the truncation by a secret shift to be correct. We arrive at $k = 72$ experimentally by computing the sizes of the shift and dot-products needed in the models we evaluate.

We ran all our benchmarks on colocated c5.9xlarge AWS machines, each of which has 36 cores, 72gb of memory, a 10gpbs link between them and sub-millisecond latency. Throughout this section, communication is measured per party and all timings include preprocessing. Our code has been published as

Variant	Accuracy		Trunc.	Active Security			
				Dishonest Maj.		Honest Maj.	
	Top-1	Top-5		$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$
V1 0.25_128	39.5%	64.4%	Prob.	1264.9	1377.8	5.3	9.0
			Exact	1864.9	1592.0	6.8	10.5
V1 0.25_160	42.8%	68.1%	Prob.	1997.4	2070.4	8.1	14.3
			Exact	2916.1	2432.8	10.6	16.9
V1 0.25_192	45.7%	70.8%	Prob.	2827.8	2875.0	11.8	20.3
			Exact	4173.9	3389.8	15.3	24.1
V1 0.25_224	48.2%	72.8%	Prob.	3825.6	3855.3	16.1	27.3
			Exact	5629.6	4574.0	20.6	32.5
V1 0.5_128	54.9%	78.1%	Prob.	2731.5	2760.4	10.9	18.5
			Exact	3950.3	3183.4	14.0	21.8
V1 0.5_160	57.2%	80.5%	Prob.	4331.5	4277.2	17.7	29.8
			Exact	6177.5	5006.9	22.3	34.7
V1 0.5_192	59.9%	82.1%	Prob.	6194.6	6026.6	25.5	42.7
			Exact	8924.5	7025.4	32.6	50.9
V1 0.5_224	61.2%	83.2%	Prob.	8446.5	8112.9	33.7	57.7
			Exact	11962.2	9143.3	43.7	68.2
V1 0.75_128	55.9%	79.1%	Prob.	4440.8	4152.6	17.4	29.3
			Exact	6203.2	4754.5	21.9	34.4
V1 0.75_160	62.4%	83.7%	Prob.	7018.5	6502.8	28.3	46.9
			Exact	9780.5	7491.2	36.1	55.1
V1 0.75_192	66.1%	86.2%	Prob.	10053.3	9145.2	40.5	68.0
			Exact	13991.2	10696.1	50.8	78.5
V1 0.75_224	66.9%	86.9%	Prob.	13634.5	12367.3	54.4	91.9
			Exact	18962.2	14370.4	69.3	107.7
V1 1.0_128	63.3%	84.1%	Prob.	6381.8	5733.0	24.9	40.9
			Exact	8797.0	6624.6	31.1	48.1
V1 1.0_160	66.9%	86.7%	Prob.	10142.0	9006.3	39.9	65.3
			Exact	13780.4	10357.2	49.6	76.8
V1 1.0_192	69.1%	88.1%	Prob.	14471.8	12778.3	57.0	95.3
			Exact	19725.0	14770.0	71.4	110.0
V1 1.0_224	70.0%	89.0%	Prob.	19691.6	17211.3	76.9	129.0
			Exact	26714.3	19910.4	96.2	151.3

Table 4.4: As the previous table, but active security.

part of MP-SPDZ [63].

Microbenchmarks

The main gain in efficiency is obtained by virtue of dot-products, or sums-of-products, being essentially free in some of the protocols we evaluate. We illustrate the efficiency of this optimization in Table 4.2.

Our micro-benchmarks are focused first and foremost on measuring the cost, in terms of time and communication, of the core operation of any CNN: the sum-of-product operation (in the MobileNets models, essentially all computations are convolutions). The top table in Table 4.2 shows the result of running a variable number of dot products each of a fixed length, and the bottom table in Table 4.2 shows the result of running a fixed number of dot products with variable length. We choose numbers that reflect realistic sizes for the

convolutions, for example, the largest convolution in the smallest MobileNetsV1 network contains some 60K dot products.

Unsurprisingly, we see a noticeable slowdown for protocols where the communication cost of dot-products depend on the number of terms. For example, the active security modulo p protocol has a runtime increase of roughly $\times 2.4$, when the number of terms is quadrupled, whereas the passive modulo 2^k only sees a $\times 1.3$ increase in cost.

Full model evaluation

We evaluate 16 pre-trained V1 MobileNet models of varying sizes. Each model is evaluated across four different dimensions:

1. Corruption threshold: We evaluate with both honest and dishonest majority, where the former uses three parties and the latter two.

2. Corruption model: Passive vs. active security.

3. Algebraic structure: We consider protocols over rings and protocols over fields, with parameters as outlined above.

4. Probabilistic vs. exact truncation.

Full end-to-end (i.e., with pre-processing) evaluation times for all models in all settings are shown in Table 4.3 and 4.4.[14]

Discussion. For the following discussion, we will mainly rely on the two graphs in Figure 4.2 and Figure 4.3. Both graphs use the 8 models from Table 4.3 and Table 4.4 with $S \in \{128, 192\}$. Figure 4.2 are evaluation times for protocols with honest majority while Figure 4.3 are protocols with dishonest majority.

As a first thing, we observe that corruption threshold is the most influential factor in terms of evaluation times. Indeed, just comparing the y-axis of Figure 4.2 with the y-axis of 4.3 shows that there is a huge difference. The overhead with respect to moving from honest majority to dishonest majority is as high as 200 times for certain configurations places (active security for $\mathbb{Z}_{2^k}$ for `V1 1.0_192`, for example). This large difference would be attributed to the expensive pre-processing that is needed in the dishonest majority case.

On the other hand, moving between different corruption models is relatively cheap. In this regard, the overhead is in fact more or less the same regardless of the threshold. I.e., moving from passive to active security only increases the inference time by a factor of between 3 and 30.

[14]The original publication contains just a single table, which were split into two for this thesis for formatting reasons.

		Passive Security			Active Security		
		Dishonest Maj.		Honest Maj.	Dishonest Maj.		Honest Maj.
	# parties	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{F}_p$
Time (s)	3	401.1	320.9	5.5	2456.8	2255.0	24.8
	4	799.6	597.4	7.4	3632.8	3063.7	36.0
	5	1332.9	959.5	15.8	4814.3	3921.0	54.7
Comm. (GB)	3	594.8	114.8	7.3	3513.8	515.2	31.8
	4	1183.0	232.2	8.2	5266.5	766.4	35.8
	5	1965.1	389.2	20.7	7018.7	1016.7	68.6

Table 4.5: Time and communication per party for computing V1 0.25_128 with probabilistic truncation.

We also observe that the choice of algebraic structure—field vs. ring—provides a performance boost in some cases. The ring-based protocols mostly outperform the field-based protocols in the passive case, while the reverse is true for active security. This is because we use homomorphic encryption with fields in this case but oblivious transfer with rings, which has a higher communication requirement. Otherwise, we attribute the difference to the fact that the ring we use is smaller than the field for security requirements. In particular, operations over $\mathbb{Z}_{2^{72}}$ can be performed by operating on only 72-bits (in particular, support for 128-bit wide types which exist in e.g., GCC can be used), while operating over $\mathbb{F}_p$ for $p \approx 2^{128}$ require multiplication of two 128-bit integers *without* overflow even when using Montgomery representation. Furthermore, we use the faster comparison proposed by Mohassel and Rindal [117] in the honest-majority setting with rings.

Finally, we observe, not surprisingly, that inference can be sped up by relying on a less precise method of truncation. For example, if we consider the first rows (model `V1 0.25_128`) in Table 4.3 and Table 4.4 we see that probabilistic truncation speeds up inference by between 80% and 15%. However, this increase in efficiency comes at the cost of a (possible) decrease in accuracy. We do not expect that this boost in efficiency will become more pronounced for deeper models, since the exact truncation protocol only depends on the size of the integers being truncated.

Scaling. For protocols that support more than three parties, Table 4.5 shows how the simplest network scales with up to five parties. Note we do not use Shamir of replicated secret sharing here for honest-majority computation. This explains why there is no ring-based protocol and the discrepancy between the results here and in Table 4.3 and Table 4.4. The number of corrupted parties is set to the maximum in the respective protocols, that is, 2, 3, 4 for dishonest majority, and 1, 1, 2 for honest majority.

		A	B	C	D	SN
Time	CrypTFlow	16	57	90	24	622
	Ours w/o TruncPrSp	23	67	122	22	2099
	Ours w/ TruncPrSp	13	18	49	15	484
Comm	CrypTFlow	1.9	6.2	15.3	2.2	187
	Ours w/o TruncPrSp	2.3	28.1	44.3	3.9	512
	Ours w/ TruncPrSp	1.1	2.6	7.0	1.1	59

Table 4.6: Time (in ms) and total communication (in MB) for SecureNN A–D and CIFAR10 SqueezeNet networks.

Special Truncation

In order to evaluate the benefit of our special truncation, we have benchmarked our implementation with and without it against CrypTFlow [109] using the SecureNN Networks A–D [142] as well as the CIFAR10 SqueezeNet examples in the CrypTFLow codebase [129]. Networks A–D are simple networks consisting of up to about ten layers using only matrix multiplication, convolution, ReLU, and max-pooling while the CIFAR10 SqueezeNet involves more than ten of each convolutions and ReLU. Table 4.6 shows our results using the same network setup as previously described. Special truncation consistently improves over CrypTFlow whereas the results without are sometimes considerably worse. The improvement is noticeable because truncation in CrypTFlow simply consists of local operations whereas we use a protocol for this. The protocol has the advantage that it does not pose restrictions on the secret value whereas the method in CrypTFlow requires that the most significant s bits of the secret are zero for a statistical security parameter s. Without special truncation, we rely on a protocol that requires k random bits to mask a k-bit value. The generation of random bits in turn requires at least k bit in communication, which makes the overall communication quadratic in k. Special truncation however has communication cost linear in k. Note also that we use comparisons as in ABY^3 [117], which is comparable to the approach in CrypTFlow in that the protocol is very specific to the security model and computation domain.

We have also implemented CrypTFlow's ImageNet examples with and without special truncation. The results can be found in Table 4.7. Note the results were obtained with the optimal number of threads for both frameworks, which is 32 for MP-SPDZ and 8 for CrypTFlow.

4.5 Conclusions

We show that it is possible to securely evaluate large and realistic networks, so called ImageNet networks, using more-or-less existing MPC protocols. Moreover, the networks we evaluate are unmodified and can be trained using standard Tensorflow or any other framework which supports the type of quantization

		SN	RN-50	DN-121
Time	CrypTFlow	10.9	26.9	37.2
	Ours w/o TruncPrSp	2.5	18.9	19.8
	Ours w/ TruncPrSp	0.6	4.7	3.7
Comm	CrypTFlow	2.6	6.9	10.5
	Ours w/o TruncPrSp	7.4	53.0	60.3
	Ours w/ TruncPrSp	0.8	3.8	4.6

Table 4.7: Time (in s) and total communication (in GB) for SqueezeNet, ResNet-50, and DenseNet-121 classification for ImageNet.

discussed (which currently includes both PyTorch and MXNet). This work thus provides a very appealing approach to secure evaluation from an end-users perspective: First, because standard MPC suffices, it is possible to choose from a wider array of threat models than previous works allow. While the passive security honest majority setting is by far the most efficient, our benchmarks still provide an interesting insight into the exact trade-off one wants secure inference against dishonest majority. Second, the fact that models directly output by Tensorflow can be evaluated *without* modification, means that model designers can remain oblivious to the secure framework. However, we also saw that choices of more specialized protocols, such as our special probabilistic truncation, can be beneficial if one wants a trade-off in terms of prediction accuracy and speed.

Chapter 5

LSS Homomorphisms and Applications to Secure Signatures, Proactive Secret Sharing and Input Certification

Diego, Aranha, Anders Dalskov, Daniel Escudero, Claudio Orlandi

Aarhus University, Denmark

Abstract. In this paper we present the concept of linear secret-sharing homomorphisms, which are linear transformations between different secret-sharing schemes defined over vector spaces over a field $\mathbb{F}$ and allow for efficient multiparty conversion from one secret-sharing scheme to the other. This concept generalizes the observation from (Smart and Talibi, IMACC 2019) and (Dalskov et al., ESORICS 2020) that moving from a secret-sharing scheme over $\mathbb{F}_p$ to a secret sharing over an elliptic curve group $\mathbb{G}$ of order p can be done non-interactively by multiplying the share unto a generator of $\mathbb{G}$. We generalize this idea an show that it can also be used to compute arbitrary bilinear maps and in particular pairings over elliptic curves.

We present several practical applications using our techniques: First we show how to securely realize the Pointcheval-Sanders signature scheme (CT-RSA 2016) in MPC. Second we present a construction for dynamic proactive secret-sharing which outperforms the current state of the art from CCS 2019. Third we present a construction for MPC input certification using digital signatures that we show experimentally to outperform the previous best solution in this area. Finally, we also show alternative ways of encoding and decoding secret-shared data to and from $\mathbb{F}_p$ and $\mathbb{G}$.

5.1 Introduction

A (t, n)-secure secret-sharing scheme allows a secret to be distributed into n shares in such a way that any set of at most t shares are independent of the secret, but any set of at least $t+1$ shares together can completely reconstruct the secret. In *linear* secret-sharing schemes (LSSS), shares of two secrets can be added together to obtain shares of the sum of the secrets. A popular example of a $(n-1, n)$-secure LSSS is additive secret sharing, whereby a secret $s \in \mathbb{F}_p$ (here $\mathbb{F}_p$ denotes integers modulo a prime p) is secret-shared by sampling uniformly random $s_1, \ldots, s_n \in \mathbb{F}_p$ subject to $s_1 + \cdots + s_n \equiv s \bmod p$. Another well-known example of a (t, n)-secure LSSS is Shamir secret sharing [135] that distributes a secret $s \in \mathbb{F}_p$ by sampling a random polynomial $f(x)$ over $\mathbb{F}_p$ of degree at most t such that $f(0) = s$, and where the i'th share is defined as $s_i = f(i)$.

Linear secret-sharing schemes are information-theoretic in nature: they do not rely on any computational assumption and therefore tend to be very efficient. Furthermore, they are widely used in multiple applications like distributed storage [76] or secure multiparty computation [49]. Linear secret-sharing schemes can be augmented with techniques from public-key cryptography, such as elliptic-curve cryptography. As an example, consider (a variant of) Feldman's scheme for verifiable secret sharing[1] [71]: To distribute a secret $s \in \mathbb{F}_p$, the dealer samples a polynomial of degree at most t such that $f(0) = s$, say $f(x) = s + r_1 x + \cdots + r_t x^t$, and sets the i-th share to be $s_i = f(i)$. On top of this, the dealer publishes $s \cdot G, r_1 \cdot G, \ldots, r_t \cdot G$, where G is a generator of an elliptic-curve group $\mathbb{G}$ of order p for which the discrete-log problem is hard. Each party can now detect if its share s_i is correct by computing $s_i \cdot G$ and checking that it equals $s \cdot G + i^1(r_1 G) + i^2(r_2 G) + \cdots + i^t(r_t G)$.

Similar approaches have also been used to instantiate polynomial commitments [101], or to securely compute ECDSA signatures [56, 138]. The key idea behind these techniques is that the group $\mathbb{G}$, having order p, is homomorphic to $\mathbb{F}_p$ as an additive group. A linear secret sharing scheme over $\mathbb{F}_p$, that satisfies some kind of "homomorphism" mod p, can therefore seen to be "compatible" with arithmetic over $\mathbb{G}$ as well. This interaction enables applications that exploit primitives that require computational assumptions, like commitments or signatures, together with efficient distributed information-theoretic techniques of linear secret sharing.

In this work we formalize and generalize the above notion by using an adequate mathematical definition of LSSS, extending it to general vector spaces, of which elliptic curves are particular cases, and using linear transformations between these vector spaces to convert from one secret-shared representation to a different one. We extend this notion and show how generic multiplication triples over $\mathbb{F}_p$ can be used to securely compute general bilinear maps, of which

[1] A verifiable secret-sharing scheme is one in which parties can verify that the dealer shared the secret correctly

bilinear *pairings* are a particular case. Our techniques neatly generalize those used in some prior work like [56, 138] and we present several applications that demonstrate how our techniques can be used to obtain protocols that outperform current state-of-the-art.

Our Contributions

The contributions made in this work are summarized here. This listing also serves as an overview of the rest of the paper.

- We introduce the concept of **linear secret-sharing homomorphism** (LSS homomorphisms) which can be seen as a generalization formalization of the idea of "putting the share in the exponent". An adequate mathematical foundation for LSS homomorphisms is presented, and we show how generic multiplication triples can be used to compute securely any bilinear map. This is done in Section 5.2.

- We demonstrate how LSS homomorphisms permits computation of scalar products, thus showing that it generalizes previously used techniques in e.g., [56, 138]. We furthermore show that it is possible to use our techniques to compute bilinear pairings over secret-shared data using any secure computation protocol. In Section 5.3 this is done, where the first part shows how to compute scalar multiplications and bilinear pairings, and where the second part shows how to instantiate our techniques with various popular secret-sharing schemes.

- To illustrate the usefulness of our LSS homomorphisms, we provide 3 applications. The first of these is a demonstration of how digital signatures can be computed and verified on secret-shared data. This is done in Section 5.4.

- Our second application demonstrates a protocol for dynamic proactive secret-sharing (PSS). This uses the digital signatures and the result is a dynamic PSS protocol with better communication complexity than the current state of the art. This is done in Section 5.5.

- Our final application is input certification. We present a method for verifying that a certain party provided input to a secure computation that was previously certified by a trusted party. We benchmark our protocol experimentally and show that it outperforms the previous best solution for input certification for any number of parties. The protocol is presented in Section 5.6, and our experiments are presented in Section 5.7.

- Our LSS Homomorphisms can be viewed as a way of encoding a value from $\mathbb{F}_p$ in $\mathbb{G}$; however they unfortunately do not permit efficient decoding.

We therefore present a method that permits both efficient encoding and decoding in MPC between $\mathbb{F}_p$ and $\mathbb{G}$. Due to space constraints, this is presented in Appendix 5.7.

Related Work

Previous works [56, 138] make use of the folklore idea of "putting the shares in the exponent" to efficiently instantiate threshold ECDSA, among other things. They approach the problem from a more practical point of view, using certain specific protocols and focusing on the application at hand, whereas our work is more general, applying to *any* linear secret-sharing scheme and also any vector space homomorphism. Furthermore, these works did not consider the case of cryptographic pairings, as these are not needed in the ECDSA algorithm.

Multiple works have addressed the problem of proactive secret-sharing. It was originally proposed in [93, 121], and several works have built on top of these techniques [18, 19, 94, 113, 134], including ours. Among these, the closest to our work is the state-of-the-art [113], which also makes use of pairing-friendly elliptic curves to ensure correctness of the transmitted message. However, a crucial difference is that in their work, a commitment scheme based on elliptic curves, coupled with the technique of "putting the share in the exponent" is used to ensure each player *individually* behaves correctly. Instead, in our work, we use elliptic curve computation on the secret rather than on the shares, which reduces the communication complexity, as shown in Section 5.5.

Finally, not many works have been devoted to the important task of input certification in MPC. For general functions, the only works we are aware of are [25, 26, 102, 146]. Among these, only [26] tackles the problem from a more general perspective, having multiple parties and different protocols. In [26], the concept of signature schemes with privacy is introduced, which are signatures that allow for an interactive protocol for verification, in such a way that the privacy of the message is preserved. The authors of [26] present constructions of this type of signatures, and use them to solve the input certification problem. However, the techniques from [26] differ from ours at a fundamental level: Their protocols first computes a commitment of the MPC inputs, and then engage in an interactive protocol for verification to check the validity of these inputs. Furthermore, these techniques are presented separately for two MPC protocols: one from [58] and one from [59]. Instead, our results apply to *any* MPC protocol based on linear secret-sharing schemes, and moreover, is much simpler and efficient as no commitments, proofs of knowledge, or special verification protocol are needed.

5.2 LSS Homomorphisms and Bilinear Maps

Let $\mathbb{F}$ be a prime field of order p. We use $a \in_R A$ to represent that a is sampled uniformly at random from the finite set A.

Linear Secret Sharing

In this section we define the notion of linear secret sharing that we will use throughout this paper. Most of the presentation here can be seen as a simplified verision of [50, Section 6.3], but it can also be regarded as a generalization since we consider arbitrary vector spaces.

Definition 1. *Let $\mathbb{F}$ be a field. A linear secret sharing scheme (LSSS) $\mathcal{S}$ over V for n players is defined by a matrix $M \in \mathbb{F}^{m\times(t+1)}$, where $m \geq n$, and a function* $\mathsf{label} : \{1, \dots, m\} \to \{1, \dots, n\}$*. We say M is the matrix for $\mathcal{S}$. We can apply* label *to the rows of M in a natural way, and we say that player $P_{\mathsf{label}(i)}$ owns the i-th row of M. For a subset A of the players, we let M_A be the matrix consisting of the rows owned by players in A.*

To secret-share a value $s \in V$, the dealer samples uniformly at random a vector $\boldsymbol{r}_s \in V^{t+1}$ such that its first entry is s, and sends to player P_i each row of $M \cdot \boldsymbol{r}_s$ owned by this player. We write $[\![s, \boldsymbol{r}_s]\!]$ for the vector of shares $M \cdot \boldsymbol{r}_s$, or simply $[\![s]\!]$ if the randomness vector $\boldsymbol{r}_s$ is not needed. Observe that the parties can obtain shares of $s_1 + s_2$ from shares of s_1 and shares of s_2 by locally adding their respective shares. We denote this by $[\![s_1 + s_2]\!] = [\![s_1]\!] + [\![s_2]\!]$.

The main properties of a secret sharing scheme are privacy and reconstruction, which are defined with respect to an access structure. In this work, and for the sake of simplicity, we consider only threshold access structures. That said, our results generalize without issue to more general access structures as well.

Definition 2. *An LSSS $\mathcal{S} = (M, \mathsf{label})$ is $(t, t+1)$-secure if the following holds:*

- *(Privacy) For all $s \in V$ and for every subset A of players with $|A| \leq t$, the distribution of $M\boldsymbol{r}_s$ is independent of s*
- *(Reconstruction) For every subset A of players with $|A| \leq t$ there is a reconstruction vector $\boldsymbol{e}_A \in \mathbb{F}^{m_A}$ such that $\boldsymbol{e}_A^\intercal(M_A\boldsymbol{r}_s) = s$ for all $s \in V$.*

LSS over Vector Spaces

Let V be a finite-dimensional $\mathbb{F}$-vector space, and let $\mathcal{S} = (M, \mathsf{label})$ be an LSSS over $\mathbb{F}$. Since V is isomorphic to $\mathbb{F}^k$ for some k, we can use the LSSS $\mathcal{S}$ to secret-share elements in V by simply sharing each one of its k components. This is formalized as follows.

Definition 3. *A linear secret-sharing scheme over a finite-dimensional $\mathbb{F}$-vector space V is simply an LSSS $\mathcal{S} = (M, \mathsf{label})$ over $\mathbb{F}$. To share a secret $v \in V$, the dealer samples uniformly at random a vector $\boldsymbol{r}_v \in V^{t+1}$ such that its first entry is v, and sends to player P_i each row of $M \cdot \boldsymbol{r}_v \in V^m$ owned by this player. Privacy properties are preserved. To reconstruct, a set of parties A with $|A| > t$ uses the reconstruction vector $\boldsymbol{e}_A$ as $\boldsymbol{e}_A^\intercal(M_A\boldsymbol{r}_v) = v$.*

As before, given $v \in V$ we use the notation $[\![v, \boldsymbol{r}_v]\!]_V$, or simply $[\![v]\!]_V$, to denote the vector in V^m of shares of v.

LSS Homomorphisms

Let U and V be two finite-dimensional $\mathbb{F}$-vector spaces, and let $\phi : V \to U$ be a vector-space homomorphism. According to the definition in Section 5.2, any given LSSS $\mathcal{S} = (M, \mathsf{label})$ over $\mathbb{F}$ can be seen as an LSSS over V or over U. However, the fact that there is a vector-space homomorphism from V to U implies that, for any $v \in V$, the parties can locally get $[\![\phi(v)]\!]_U$ from $[\![v]\!]_V$. We formalize this below.

Definition 4. *Let U and V be two finite-dimensional $\mathbb{F}$-vector spaces, and let $\phi : V \to U$ be a vector-space homomorphism. Let $\mathcal{S} = (M, \mathsf{label})$ be an LSSS over V. We say that the pair $(\mathcal{S}, \phi)$ is a linear secret-sharing homomorphism.*

The following simple proposition illustrates the value of considering LSS homomorphisms.

Proposition 2. *Let U and V be two finite-dimensional $\mathbb{F}$-vector spaces, and let $(\mathcal{S}, \phi)$ be a LSS homomorphism from U to V. Given $v \in V$ and $[\![v, \boldsymbol{r}_v]\!]_V$, applying ϕ to each share leads to $[\![\phi(v), \phi(\boldsymbol{r}_v)]\!]_U$.*[2]

Proof. Observe that $\phi\left([\![v, \boldsymbol{r}_v]\!]_V\right) = \phi(M\boldsymbol{r}_v) = M\phi(\boldsymbol{r}_v) = [\![\phi(v), \phi(\boldsymbol{r}_v)]\!]_U$. □
□

LSSS with Bilinear Maps

In Section 5.2 we saw how the parties could locally convert from sharings in one vector space to another vector space, provided there is a linear transformation between the two. The goal of this section is to extend this to the case of bilinear maps. More precisely, let U, V, W be $\mathbb{F}$-vector spaces of dimension d,[3] and let $\mathcal{S} = (M, \mathsf{label})$ be an LSSS over $\mathbb{F}$. From Section 5.2, $\mathcal{S}$ is also an LSSS over U, V and W. Let $\phi : U \times V \to W$ be a bilinear map, that is, the functions $\phi(\cdot, v)$ for $v \in V$ and $\phi(u, \cdot)$ for $u \in U$ are linear.

We show how the parties can obtain $[\![\phi(u, v)]\!]_W$ from $[\![u]\!]_U$ and $[\![v]\!]_V$, for $u \in U$ and $v \in V$. Unlike the case of a linear transformation, this operation requires communication among the parties. Intuitively, this is achieved by using a generalization of "multiplication triples" to the context of bilinear maps. At a high level, the parties preprocess "bilinear triples" $([\![\alpha]\!]_U, [\![\beta]\!]_V, [\![\phi(\alpha, \beta)]\!]_W)$ where $\alpha \in U$ and $\beta \in V$ are uniformly random, open $\delta = u - \alpha$ and $\epsilon = v - \beta$,

[2] We extend the definition of ϕ to operate on vectors over V pointwise.

[3] It is not necessary for these spaces to have the same dimension, but we assume this for simplicity in the notation.

and compute $[\![\phi(u,v)]\!]_W$ as

$$\begin{aligned}\phi(\delta,\epsilon)+\phi(\delta,[\![\beta]\!]_V)+\phi([\![\alpha]\!]_U,\epsilon)+[\![\phi(\alpha,\beta)]\!]_W &= [\![\phi\left(\delta+\alpha,\epsilon+\beta\right)]\!]_W \\ &= [\![\phi\left(u,v\right)]\!]_W.\end{aligned}$$

Appendix 5.7 formalizes this intuition and defines a protocol Π_{bilinear} parameterized by the map ϕ, which takes as input $[\![u]\!]_U$, $[\![v]\!]_V$ and outputs $[\![w]\!]_W$ with $w=\phi(u,v)$.

5.3 Instantiations

In the previous section we developed a theory for LSS homomorphisms and secure computation for bilinear maps based on an arbitrary linear secret sharing scheme and an arbitrary linear transformation between vector spaces. In this section we instantiate the vector spaces with elliptic curves, and the bilinear maps with cryptographic pairings, which allows us to securely evaluate cryptographic primitives based on elliptic curves and pairings. Moreover, we provide an exact description of how to instantiate the secret-sharing scheme using different types of sharings. In particular additive secret-sharing with MACs (as in SPDZ [59]); Shamir secret-sharing used in honest majority protocols [58]; and replicated secret-sharing which is particularly efficient for the special case of 3 parties and 1 corruption [8].

Instantiating the Vector Spaces with Elliptic Curves

Let $\mathbb{G}$ be an elliptic curve group of order a prime p, which in particular means that $\mathbb{G}$ is an $\mathbb{F}$-vector space, and let G be a generator of $\mathbb{G}$. Consider the isomorphism $\phi:\mathbb{F}\to\mathbb{G}$ given by $x\mapsto x\cdot G$. Let $\mathcal{S}=(M,\mathsf{label})$ be an LSSS over $\mathbb{F}$. Given what we have seen so far, $\mathcal{S}$ can be seen as an LSSS over $\mathbb{G}$. To secret-share a curve point $P\in\mathbb{G}$, the dealer samples random points $(P_1,\dots,P_t)$, computes $(Q_1,\dots,Q_m)^\intercal=M\cdot(P,P_1,\dots,P_t)^\intercal\in\mathbb{G}^m$, and sends Q_i to party $P_{\mathsf{label}(i)}$. Furthermore, if $s\in\mathbb{F}$ is secret shared as $[\![s]\!]$, the LSS homomorphism property applied to ϕ implies that each party can locally multiply its share by the generator G to obtain $[\![s\cdot G]\!]_{\mathbb{G}}$.

Now, consider the scalar multiplication map $f:\mathbb{F}\times\mathbb{G}\to\mathbb{G}$ given by $f: x,P\mapsto x\cdot P$. Using Π_{Bilinear} with f we can obtain the protocol $\Pi_{\mathsf{ScalarMult}}$ (more precisely, $\Pi_{\mathsf{ScalarMult}}$ is a special case of Π_{Bilinear} when the LSS homomorphism is f and the dimensions of the inputs are 1), described below, which computes a scalar multiplication between a scalar and point when both scalar and point are secret-shared. We remark that this protocol was presented in [138] and as such our presentation here can be considered as illustrating that Π_{Bilinear} generalizes the techniques in their work. We assume access to a triple preprocessing functionality $\mathcal{F}_{\mathsf{MultTriple}}$ that produces $([\![a]\!],[\![b]\!],[\![a\cdot b]\!])$, where $a,b\in\mathbb{F}$ are uniformly random.

Protocol 1 Protocol $\Pi_{\mathsf{ScalarMult}}$

Inputs: $[\![x]\!]$ and $[\![P]\!]_{\mathbb{G}}$

Outputs: $[\![x \cdot P]\!]_{\mathbb{G}}$

Offline Phase:

1. Parties call $([\![a]\!], [\![b]\!], [\![a \cdot b]\!]) \leftarrow \mathcal{F}_{\mathsf{MultTriple}}$.
2. Parties use the LSS homomorphism $x \mapsto x \cdot G$ for a generator G of $\mathbb{G}$ to compute $[\![B]\!]_{\mathbb{G}} = [\![b]\!] \cdot G$ and $[\![C]\!] = [\![a \cdot b]\!] \cdot G$.

Online Phase:

1. Parties open $d \leftarrow [\![x]\!] - [\![a]\!]$ and $Q \leftarrow [\![P]\!]_{\mathbb{G}} - [\![B]\!]_{\mathbb{G}}$.
2. Using the LSS homomorphism, parties compute $[\![E]\!]_{\mathbb{G}} = [\![a]\!] \cdot Q$ and $[\![F]\!]_{\mathbb{G}} = d \cdot [\![B]\!]_{\mathbb{G}}$.
3. Parties compute locally $[\![x \cdot P]\!]_{\mathbb{G}} = [\![E]\!]_{\mathbb{G}} + [\![F]\!]_{\mathbb{G}} + d \cdot Q + [\![C]\!]_{\mathbb{G}}$.

Bilinear Pairings

Consider $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ elliptic curve groups of order a prime p. As usual in the field of pairing-based cryptography, we use additive notation for the groups $\mathbb{G}_1, \mathbb{G}_2$, and multiplicative notation for $\mathbb{G}_T$. We denote by $0_{\mathbb{G}_1}, 0_{\mathbb{G}_2}$ and $1_{\mathbb{G}_T}$ the identities of $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$, respectively. Consider a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ satisfying:

1. For all $G \in \mathbb{G}_1, H \in \mathbb{G}_2$ and $a, b \in \mathbb{F}$, $e(aG, bH) = e(G, H)^{ab}$.
2. For $P_1 \in \mathbb{G}_1$, $P_2 \in \mathbb{G}_2$ with $P_1 \neq 0$, $P_2 \neq 0$, $e(P_1, P_2) \neq 1$.
3. The map e can be computed efficiently.

This notation will be used for the rest of the paper. In the context of Section 5.2, the groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ can be viewed as $\mathbb{F}$-vector spaces of dimension 1, so we can apply the techniques presented there to compute $[\![e(P_1, P_2)]\!]_{\mathbb{G}_T}$ from $[\![P_1]\!]_{\mathbb{G}_1}$ and $[\![P_2]\!]_{\mathbb{G}_2}$. We summarize the resulting protocol below. We let G_1 and G_2 denote generators of $\mathbb{G}_1$ and $\mathbb{G}_2$, respectively.

Protocol 2 Protocol Π_{Pairing}

Inputs: $[\![P_1]\!]_{\mathbb{G}_1}$ and $[\![P_2]\!]_{\mathbb{G}_2}$.

Output: $[\![e(P_1, P_2)]\!]_{\mathbb{G}_T}$.

Offline Phase:

1. The parties call $(\llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket a \cdot b \rrbracket) \leftarrow \mathcal{F}_{\mathsf{MultTriple}}$.

2. The parties use the LSS homomorphisms $x \mapsto x \cdot G_1$ and $x \mapsto x \cdot G_2$ to locally compute $\llbracket Q_1 \rrbracket_{\mathbb{G}_1} = \llbracket a \rrbracket \cdot G_1$ and $\llbracket Q_2 \rrbracket_{\mathbb{G}_2} = \llbracket b \rrbracket \cdot G_2$, respectively.

3. Using the LSS homomorphism $x \mapsto e(G_1, G_2)^x$, the parties compute $\llbracket e(Q_1, Q_2) \rrbracket = \llbracket e(a \cdot G_1, b \cdot G_2) \rrbracket_{\mathbb{G}_T} \leftarrow e(G_1, G_2)^{\llbracket ab \rrbracket}$

Online Phase:

1. The parties open $D_1 \leftarrow \llbracket P_1 \rrbracket_{\mathbb{G}_1} - \llbracket Q_1 \rrbracket_{\mathbb{G}_1}$ and $D_2 \leftarrow \llbracket P_2 \rrbracket_{\mathbb{G}_2} - \llbracket Q_2 \rrbracket_{\mathbb{G}_2}$

2. The parties use the LSS homomorphism $e(Q_1, \cdot)$ to compute $\llbracket e(D_1, Q_2) \rrbracket_{\mathbb{G}_T} \leftarrow e(D_1, \llbracket Q_2 \rrbracket_{\mathbb{G}_1})$, and similarly they use the LSS homomorphism $e(\cdot, D_2)$ to compute $\llbracket e(Q_1, D_2) \rrbracket_{\mathbb{G}_T} \leftarrow e(\llbracket Q_1 \rrbracket_{\mathbb{G}_1}, D_2)$.

3. The parties compute locally and output $\llbracket e(P_1, P_2) \rrbracket_{\mathbb{G}_T} = e(D_1, D_2) \cdot \llbracket e(D_1, Q_2) \rrbracket_{\mathbb{G}_T} \cdot \llbracket e(Q_1, D_2) \rrbracket_{\mathbb{G}_T} \cdot \llbracket e(Q_1, Q_2) \rrbracket_{\mathbb{G}_T}$.

Instantiating the Secret Sharing Schemes

We present multiple linear secret-sharing schemes, together with their instantiations over elliptic curves. We remark that this is for the sake of concreteness, but they can be instantiated over any finite-dimensional vector space V over $\mathbb{F}$.

Additive SS. In this scheme each party P_i gets a uniformly random value $r_i \in \mathbb{F}$ subject to $\sum_{i=1}^{n} r_i = s$, where $s \in \mathbb{F}$ is the secret. This scheme is $(n-1, n)$-secure. Let us denote additive secret sharing of s by $\llbracket s \rrbracket^{\mathsf{add}}$ and, abusing notation, we write $\llbracket s \rrbracket^{\mathsf{add}} = (r_1, \ldots, r_n)$, where each r_i is the share of party P_i. Given an elliptic curve group $\mathbb{G}$ of order p with generator G, the parties can obtain shares of $s \cdot G$ by locally multiplying the generator G by their share r_i; that is, $\llbracket s \cdot G \rrbracket^{\mathsf{add}} = (r_1 \cdot G, \ldots, r_n \cdot G)$.

In the scheme above, at reconstruction time, a maliciously corrupt party can lie about its share, causing the reconstructed value to be incorrect. To help solve this issue, actively secure protocols in the dishonest majority setting share a secret s as $\llbracket s \rrbracket^{\mathsf{add}}$, together with $\llbracket r \cdot s \rrbracket^{\mathsf{add}}$, where r is a *global* uniformly random value that is also shared as $\llbracket r \rrbracket^{\mathsf{add}}$. We denote this by $\llbracket s \rrbracket^{\mathsf{add}*}$. At reconstruction time, the adversary may open $\llbracket s \rrbracket^{\mathsf{add}}$ to $s + \delta$ where δ is some error known to the adversary. To ensure that $\delta = 0$ (i.e., the correct value was opened), the parties compute $(s+\delta)\llbracket r \rrbracket^{\mathsf{add}} - \llbracket r \cdot s \rrbracket^{\mathsf{add}}$, open this value, and check it equals 0. It can be easily shown that, if $\delta \neq 0$, this check passes with probability at most $1/|\mathbb{F}|$.

The same check can be performed over $\mathbb{G}$: The sharings $[\![s \cdot G]\!]_{\mathbb{G}}^{\mathsf{add}}$ are accompanied by $[\![r \cdot s \cdot G]\!]_{\mathbb{G}}^{\mathsf{add}}$, where r is a *global* uniformly random value that is also shared as $[\![r]\!]^{\mathsf{add}}$. At reconstruction time $[\![s \cdot G]\!]_{\mathbb{G}}^{\mathsf{add}}$ can be opened to $(s+\delta) \cdot G$, and to ensure $\delta = 0$ the parties open $[\![r]\!]_{\mathbb{G}}^{\mathsf{add}} \cdot (s+\delta) \cdot G - [\![r \cdot s \cdot G]\!]_{\mathbb{G}}^{\mathsf{add}}$ and check that this point is the identity. It is easy to see that, like in the case over $\mathbb{F}$, the check passes with probability at most $1/|\mathbb{F}|$ if $\delta \neq 0$. We denote this "robust" sharing of $s \cdot G$ by $[\![s \cdot G]\!]_{\mathbb{G}}^{\mathsf{add}*}$.

Shamir SS. Let $\mathbb{F}_{\leq d}[x]$ be the ring of polynomials over $\mathbb{F}$ of degree at most d. Consider a setting with n parties, and let $0 < t < n$. In this scheme each party P_i gets $f(i)$ where $f(x) \in_R \mathbb{F}_{\leq t}[x]$. We denote $[\![s]\!]_{\mathbb{F}}^{\mathsf{shm}} = (f(1), \dots, f(n))$. Recall that s can be reconstructed from t shares by computing $s = \sum_{i=1}^{t+1} \lambda_i s_i$ where s_i is the i'th share, and where λ_i is the i'th Lagrange coefficient. This works as well when reconstructing $[\![s \cdot G]\!]_{\mathbb{G}}$ since $s \cdot G = \sum_{i=1}^{t+1} \lambda_i (s_i \cdot G) = (\sum_{i=1}^{t+1} \lambda_i s_i) \cdot G$. In Section 5.7 in the Appendix we present a more detailed description of this scheme for any vector space, together with protocols for instantiating a generalized version of the functionality $\mathcal{F}_{\mathsf{DotProduct}}$ defined below.

Replicated SS. This is a $(1, 2)$-secure LSSS for 3 parties. In this scheme each party P_i gets (r_i, r_{i+1}) with $s = r_1 + r_2 + r_3$, and where $s \in \mathbb{F}$ is the secret. (We interpret $r_4 = r_1$, i.e., indices "wrap" modulo 3.) Reconstructing, as well as active security, follows from the same arguments as presented for case of additive secret-sharing.

Primitives. For the rest of the paper, we will rely on several secure computation functionalities. We list them here in brief. Also, for a functionality/protocol $\mathcal{F}_{\mathsf{abc}}/\Pi_{\mathsf{abc}}$, we denote by $\mathcal{C}_{\mathsf{abc}}$ its total communication cost, in bits.

- $\mathcal{F}_{\mathsf{MultTriple}}$ outputs a triple $([\![a]\!], [\![b]\!], [\![c]\!])$ where $c = ab$.
- $\mathcal{F}_{\mathsf{DotProduct}*}$ takes as input $([\![x_i]\!])_{i=1}^{L}$ and $([\![y_i]\!])_{i=1}^{L}$, and produces $[\![z + \delta]\!]$, where $z = \sum_{\ell=1}^{L} \phi(x_\ell y_\ell)$ and $\delta \in \mathbb{F}$ is an error known by the adversary. $\mathcal{F}_{\mathsf{DotProduct}}$ is similar, except it does not accept such error.
- $\mathcal{F}_{\mathsf{Mult}}$ takes two inputs $[\![x]\!]$ and $[\![y]\!]$, and outputs $[\![w]\!]$ where $w = xy$. $\mathcal{F}_{\mathsf{Mult}}$ is a particular case of $\mathcal{F}_{\mathsf{DotProduct}}$ for $L = 1$.
- $\mathcal{F}_{\mathsf{Rand}}(K)$ outputs $[\![x]\!]$ where $x \in K$, where K is a $\mathbb{F}$-vector space. Notice that it is enough to have a functionality which samples a secret-shared field element; to get a secret point, parties can locally apply an appropriate LSS homomorphism to obtain a secret-shared group element.
- $\mathcal{F}_{\mathsf{Coin}}(K)$ outputs a uniformly random $s \in K$ to all parties.

5.4 Digital Signatures in MPC

In this section we show how our techniques can be used to securely sign and verify messages that are secret shared, using keys that are similarly secret-shared. More precisely, we present here three protocols: First, a key generation protocol Π_{Keygen} for generating $(\mathsf{pk}, [\![\mathsf{sk}]\!])$ securely where pk is a public key and $[\![\mathsf{sk}]\!]$ a secret-shared private key. Second, a signing protocol Π_{Sign} protocol that on input a secret shared message $[\![m]\!]$ and $[\![sk]\!]$ output from Π_{Keygen} outputs $[\![\sigma]\!]$ where σ is a signature on m under sk. Finally, we present a verification protocol Π_{Verify} which on input $[\![m]\!]$, $[\![\sigma]\!]$ and pk outputs $[\![b]\!]$ where b is a value indicating whether or not σ is a valid signature on m under the private key corresponding to the public key pk.

We choose to use the signature scheme [126] by Pointcheval and Sanders (henceforth PS) as our starting point. The primary reason for choosing the PS scheme is that signatures are short and independent of the message length, and that messages do not need to be hashed prior to signing. Interestingly, computing PS signatures securely leads to a number of optimizations that are made possible since e.g., the secret key is not known by any party.

The PS Signature Scheme

The PS signature scheme signs a vector of messages $\vec{m} \in \mathbb{F}^r$ as follows (we present the multi-message variant here):

- $\mathsf{Setup}(1^\lambda)$: Output $pp \leftarrow (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, a type-3 pairing.
- $\mathsf{Keygen}(pp)$: Select random $H \leftarrow \mathbb{G}_2$ and $(x, y_1, \ldots, y_r) \leftarrow \mathbb{F}^{r+1}$. Compute $(X, Y_1, \ldots, Y_r) = (xH, y_1H, \ldots, y_rH)$ set $\mathsf{sk} = (x, y_1, \ldots, y_r)$ and $\mathsf{pk} = (H, X, Y_1, \ldots, Y_r)$.
- $\mathsf{Sign}(\mathsf{sk}, \vec{m})$: Select random $G \leftarrow \mathbb{G}_1 \setminus \{0\}$ and output the signature $\sigma = (G, (x + \sum_{i=1}^{r} m_i y_i) \cdot G)$.
- $\mathsf{Verify}(\mathsf{pk}, \vec{m}, \sigma)$: Parse σ as (σ_1, σ_2). If $\sigma_1 \neq 0$ and $e(\sigma_1, X + \sum m_i Y_i) = e(\sigma_2, H)$ output 1. Otherwise output 0.

The remainder of this section will focus on how to instantiate the PS signature scheme securely.

Distributed PS Signatures

The Π_{Keygen} protocol presented below shows how to generate keys suitable for signing messages of r blocks. The protocol proceeds as follows: parties invoke $\mathcal{F}_{\mathsf{Coin}}$ and $\mathcal{F}_{\mathsf{Rand}}$ a suitable number of times to generate the private key and then use an appropriate LSS homomorphism to compute the public key.

Protocol 3 Protocol Π_{Keygen}

Inputs: $pp = (p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$, r.

Outputs: $(\mathsf{pk}, [\![\mathsf{sk}]\!])$.

1. Parties invoke $\mathcal{F}_{\mathsf{Coin}}(\mathbb{G}_2)$ to obtain H, and invoke $\mathcal{F}_{\mathsf{Rand}}(\mathbb{F})$ a total of $r+1$ times to obtain $([\![x]\!], [\![y_1]\!], \ldots, [\![y_r]\!])$.
2. Let $\phi_2 : \mathbb{F} \to \mathbb{G}_2$ be LSS-homomorphism given by $\phi_2 : x \mapsto xH$. Using ϕ_2, compute $[\![X]\!]_{\mathbb{G}_2} = \phi_2([\![x]\!])$ and $[\![Y_i]\!]_{\mathbb{G}_2} = \phi_2([\![y_i]\!])$ for $i = 1, \ldots, r$.
3. Parties open $X \leftarrow [\![X]\!]_{\mathbb{G}_2}$ and $Y_i \leftarrow [\![y_i]\!]_{\mathbb{G}_2}$ for $i = 1, \ldots, r$. Output the pair $(\mathsf{pk}, [\![\mathsf{sk}]\!])$ where $\mathsf{pk} = (H, X, Y_1, \ldots, Y_r)$ and $[\![\mathsf{sk}]\!] = ([\![x]\!], [\![y_1]\!], \ldots, [\![y_r]\!])$.

The communication complexity of Π_{Keygen} is $\mathcal{C}_{\mathsf{Keygen}} = \mathcal{C}_{\mathsf{Coin}}(1) + \mathcal{C}_{\mathsf{Rand}}(r+1) + \mathcal{C}_{\mathsf{Open}}(r+1)$ field elements.

Next up is computing Sign on secret-shared inputs given the tools we have described so far. The Π_{Sign} protocol below outputs a signature $(\sigma_1, [\![\sigma_2]\!]_{\mathbb{G}_1})$. The reasons for keeping σ_1 is (1) that it simplifies things when we use this later, and (2) makes signing more efficient. If, however, σ_1 cannot be revealed then Π_{Pairing} is needed for step 3.

Protocol 4 Protocol Π_{Sign}

Inputs: $[\![\mathsf{sk}]\!] = ([\![x]\!], [\![y_1]\!], \ldots, [\![y_r]\!])$, $[\![\vec{m}]\!] = ([\![m_1]\!], \ldots, [\![m_r]\!])$.

Outputs: $[\![\sigma]\!]$

1. Parties obtain $\sigma_1 \in_R \mathbb{G}_1$ by invoking $\mathcal{F}_{\mathsf{Coin}}(\mathbb{G}_1)$. If $\sigma_1 = 0$, repeat this step.
2. Parties invoke $[\![z]\!] \leftarrow \mathcal{F}_{\mathsf{DotProduct}}\left(([\![y_i]\!])_{i=1}^r, ([\![m_i]\!])_{i=1}^r\right)$ and then compute $[\![w]\!] = [\![x]\!] + [\![z]\!]$.
3. Parties use the LSS homomorphism $x \mapsto x \cdot \sigma_1$ to compute locally $[\![\sigma_2]\!]_{\mathbb{G}_1} \leftarrow \Pi_{\mathsf{ScalarMult}}([\![w]\!], \sigma_1)$.
4. Output $(\sigma_1, [\![\sigma_2]\!]_{\mathbb{G}_1})$.

Protocol Π_{Sign} produces a correct signature with communication complexity $\mathcal{C}_{\mathsf{Coin}}(1) + \mathcal{C}_{\mathsf{DotProduct}}(r)$. Observe that, if the secret-sharing scheme is instan-

tiated either with Shamir or Replicated secret sharing, the communication complexity becomes independent of r.

Remark 1. *It is possible to replace $\mathcal{F}_{\mathsf{DotProduct}}$ with $\mathcal{F}_{\mathsf{DotProduct}*}$, which has the effect that the output would be $(\sigma_1, [\![\sigma_2 + \delta]\!]_{\mathbb{G}_1})$ where δ is an error introduced by the adversary. For the application in Section 5.5 this is acceptable, and thus desirable as $\mathcal{F}_{\mathsf{DotProduct}*}$ is often more efficient.*

Finally, we show verification. The verification protocol Π_{Verify} ouputs a secret-shared $\mathbb{G}_T$ element $[\![b]\!]_{\mathbb{G}_T}$ where $b = 1_{\mathbb{G}_T}$ if an only if the signature was valid. While this is not a bit, it nevertheless carries the same information. Below the signature we verify is $(\sigma_1, [\![\sigma_2]\!]_{\mathbb{G}_1})$, however if this is not the case (in particular, if σ_1 is secret-shared) then Π_{Pairing} is needed in step 4.

Protocol 5 Protocol Π_{Verify}

Inputs: $\mathsf{pk} = (H, X, Y_1, \ldots, Y_r)$, $[\![\vec{m}]\!] = ([\![m_i]\!])_{i=1}^r$, $\sigma = (\sigma_1, [\![\sigma_2]\!]_{\mathbb{G}_1})$.

Outputs: $[\![b]\!]_{\mathbb{G}_T} = [\![1_{\mathbb{G}_T}]\!]$ if $\mathsf{Verify}(\mathsf{pk}, \vec{m}, \sigma) = 0$ and a random value otherwise.

1. If $\sigma_1 = 0$ then output $[\![\mu]\!]_{\mathbb{G}_T} \leftarrow \mathcal{F}_{\mathsf{Rand}}(\mathbb{G}_T)$.
2. Compute $[\![\alpha]\!]_{\mathbb{G}_T} = e([\![\sigma_2]\!], H)$ using the LSS Homomorphism $x \mapsto xH$.
3. Locally compute $[\![\beta]\!]_{\mathbb{G}_T} = e(\sigma_1, X + \sum_{i=1}^r [\![m_i]\!] Y_i)$ using LSS homomorphisms.
4. Output $[\![b]\!]_{\mathbb{G}_T} \leftarrow \Pi_{\mathsf{ScalarMult}}([\![\rho]\!], [\![\alpha]\!]_{\mathbb{G}_T} / [\![\beta]\!]_{\mathbb{G}_T})$ where $[\![\rho]\!]$ was obtained by invoking $\mathcal{F}_{\mathsf{Rand}}$.

The communication complexity of the Π_{Verify} protocol is $\mathcal{C}_{\mathsf{Rand}}(1)+\mathcal{C}_{\mathsf{ScalarMult}}(1)$. We now argue security.

Lemma 6. *Protocol Π_{Verify} outputs a secret-sharing of 1 if $\sigma = (\sigma_1, [\![\sigma_2]\!]_{\mathbb{G}_1})$ is a valid signature on $[\![\vec{m}]\!]$ with public key pk, otherwise the protocol outputs a secret-sharing of a uniformly random element.*

Lemma 6 is proven in Appendix 5.7.

5.5 Applications to Proactive Secret Sharing

Secret-sharing allows one to distribute a secret such that an adversary with only access to some subset of the shares cannot learn anything about the secret. However as time passes it becomes harder to argue that no leakage

beyond this subset happens, and thus that the secret remains hidden from the adversary. Proactive Secret-sharing (PSS) deals with this problem by periodically "refreshing" (or *proactivizing*) shares such that shares between two stages become "incompatible".

For the special case of *dynamic PSS* (a PSS scheme is dynamic if the number of parties and threshold can change between each proactivization), CHURP is presented in [113]. In a nutshell, CHURP first performs an optimistic proactivization and, if cheating is detected, falls back to a slower method that is able to detect cheaters.

In what follows we show how to use the protocols for signatures developed in Section 5.4 to obtain a conceptually simple and efficient dynamic PSS with abort. We first develop a highly efficient protocol for proactivizing a secret that guarantees privacy, but allows the adversary to tamper with the transmitted secret. Then, we use our signatures to transmit a signature on the secret, that can be checked by the receiving committee. In this way, due to the unforgeability properties of the signature scheme, an adversary cannot make the receiving committee accept an incorrectly transmitted message. This construction leads to a 9-fold improvement in terms of communication with respect to the optimistic protocol from [113].

Proactive Secret Sharing

We present here the definitions of proactive secret sharing, or PSS for short. We remark that our goal is not to provide formal definitions of these properties but rather a high level description of what a PSS scheme is, so that we can present in a clear manner our optimizations to the work of [113].

In a PSS scheme a set of n parties have shares of a secret $[\![s]\!] = (s_1, \ldots, s_n)$ with threshold t. At a given stage, a proactivization mechanism is executed, from which the parties obtain $[\![s']\!] = (s'_1, \ldots, s'_n)$. A PSS scheme satisfies:

- *(Correctness).* It must hold that $s = s'$
- *(Privacy).* An adversary corrupting a set of at most t parties before the proactivization, and also a (potentially different) set of at most t parties after the proactivization, cannot learn anything about the secret s.

The PSS schemes we consider in this work are *dynamic* in that the set of parties holding the secret before the proactivization step may be different than the set of parties holding the secret afterwards.

Partial PSS

In what follows we denote by $C = \{P_i\}_{i=1}^n$ and $C' = \{P'_i\}_{i=1}^n$ the old a new committees, respectively. Furthermore, we denote $U = \{P_i\}_{i=1}^{t+1}$ and $U' = \{P'_i\}_{i=1}^{t+1}$. We consider Shamir secret-sharing, as defined in Section 5.3

with threshold $t < n/2$. This ensures that the parties cannot modify their shares without resulting in an error, thanks to error-detection, as discussed in Section 5.7 in the appendix. Our protocol $\Pi_{\mathsf{PartialPSS}}$ is inspired by the protocol from [19], except that, since we do not require the transmitted message to be correct, we can remove most of the bottlenecks like the use of hyper-invertible matrices or consistency checks to ensure parties send shares consistently.

Protocol 6 Protocol $\Pi_{\mathsf{PartialPSS}}([\![s]\!]^C)$

Inputs A shared value $[\![s]\!]^C = (s_1, \ldots, s_n)$ among a committee C.

Output: Either a consistently shared value $[\![s']\!]^{C'}$ or abort. If all parties behave honestly then $s' = s$.

1. Each $P_i \in C$ samples $s_{i1}, \ldots, s_{i,t+1} \in_R \mathbb{F}$ such that $s_i = \sum_{j=1}^{t+1} s_{ij}$ and sends s_{ij} to P_j for $j = 1, \ldots, t+1$.
2. Each $P_i \in U$ samples $r_{ki} \in_R \mathbb{F}$ for $k = 1, \ldots, t$, and sets $r_{0,i} = 0$.
3. Each $P_i \in U$ sets $a_{ij} = s_{ji} + \sum_{k=0}^{t} r_{ki} \cdot j^k$ and sends a_{ij} to P'_j, for each $j = 1, \ldots, n$.
4. Each $P'_j \in C'$ sets $s'_j := \sum_{i=1}^{t+1} a_{ij}$.
5. The parties in C' output the shares $(s'_1, \ldots, s'_n)$.

Theorem 2. *Protocol $\Pi_{\mathsf{PartialPSS}}$ satisfies the following properties.*

1. *The resulting sharings are consistent. Furthermore, if all the parties behave honestly, then the underlying secret is the same as provided as input.*
2. *An adversary simultaneously controlling t parties in C and t parties in C' does not learn anything about the secret input s.*

A proof of Theorem 2 can be found in Appendix 5.7.

$\Pi_{\mathsf{PartialPSS}}$ can be extended to proactivize shares $[\![\alpha]\!]_{\mathbb{G}}^{C}$, where $\mathbb{G}$ is an elliptic curve group by running the same protocol "in the exponent". More formally, the LSS homomorphism $x \mapsto x \cdot G$, where G is a generator of $\mathbb{G}$, is used.

Communication Complexity. $\Pi_{\mathsf{PartialPSS}}$ communicates a total of $n(n+1)$ field elements.

Simple and Efficient PSS with Abort

The protocol $\Pi_{\mathsf{PartialPSS}}$ presented in the previous section guarantees privacy and consistency of the new sharings, but it does not satify the main property of a PSS, which is guaranteeing that the secret remains the same. More precisely, a malicious party may disrupt the output as $[\![s+\gamma]\!]^{C'} \leftarrow \Pi_{\mathsf{PartialPSS}}([\![s]\!]^C)$, where γ is some value known by the adversary. This is of course not ideal, but it can be fixed by making use of the signature protocols proposed in Section 5.4. In a nutshell, the committee C uses $\Pi_{\mathsf{PartialPSS}}$ to send to C' not only the secret s, but also a signature on this secret using a secret-key shared by C. Then, upon receiving shares of the message-signature pair, the parties in C' proceed to verifying this pair securely using C's public key, and if this check passes then it can be guaranteed that the message was correct, since the adversary cannot produce a valid message-signature pair for a new message.

The protocol is presented more formally in Protocol Π_{PSS} below.

Protocol 7 Protocol $\Pi_{\mathsf{PSS}}([\![s]\!]^C)$

Inputs A shared value $[\![s]\!]^C = (s_1, \ldots, s_n)$ among a committee C.

Output: Consistent shares $[\![s]\!]^{C'}$ or abort.

Setup: Parties in C have a shared secret-key $[\![\mathsf{sk}_C]\!]^C$, and its corresponding public key pk_C is known by the parties in C'.[4]

1. Parties in C call $(\sigma_1, [\![\sigma_2]\!]^C) \leftarrow \Pi_{\mathsf{Sign}}([\![\mathsf{sk}_C]\!]^C, [\![s]\!]^C)$.
2. Parties in $C \cup C'$ call $[\![s']\!]^{C'} \leftarrow \Pi_{\mathsf{PartialPSS}}([\![s]\!]^C)$ and $[\![\sigma_2']\!]^{C'} \leftarrow \Pi_{\mathsf{PartialPSS}}([\![\sigma_2]\!]^C)$.
3. $P_1, \ldots, P_{t+1}$ all send σ_1 to the parties in C'. If some party in $P_j \in C'$ receives two different σ_1 from two different parties, then the parties abort.
4. Parties in C' call $v \leftarrow \Pi_{\mathsf{Verify}}([\![s']\!]^{C'}, (\sigma_1, [\![\sigma_2']\!]^{C'}), \mathsf{pk}_C)$. If $v = 0$ then the parties in C' output $[\![s']\!]^{C'}$. Else, they abort.

Intuitively, the protocol guarantees that the parties do not abort if and only if the message is transmitted correctly. This follows from the unforgeability of the signature scheme: If an adversary can cause the parties to accept with a wrong message/signature pair, then this would constitute a forged signature. The fact that privacy is maintained regardless of whether the parties abort or not is more subtle, but essentially follows from the fact that decision to abort can be shown to *independent* of the secret (thus ruling out a selective failure

attack). Put differently, a decision depends only on the error introduced by the adversary which is independent of the secret.

We summarize these properties in Theorem 3 below which we prove in Appendix 5.7. In our proof we do not reduce to the unforgeability of the signature scheme, but instead to a hard problem over elliptic curves directly. This is easier and cleaner in our particular setting, given that the signatures are produced and checked within the same protocol. The computational problem we reduce the security of Protocol Π_{PSS} to is the following, which can be seen as a natural variant of Computational Diffie-Hellman (CDH) problem over $\mathbb{G}_1$.

Definition 5 (co-CDH assumption). *Let $G \in \mathbb{G}_1$ and $G' \in \mathbb{G}_2$ be generators. Given (G, G', aG, bG') for $a, b, \in_R \mathbb{F}$, an adversary cannot efficiently find $(ab)G$.*

With this assumption at hand, which is assumed to hold for certain choices of pairing settings (see [73]), we can prove the following about the security of Π_{PSS}.

Theorem 3. *Protocol Π_{PSS} instantiates the PSS-with-abort functionality described in Section 5.5, that is, if the parties do not abort in the protocol Π_{PSS}, then the parties in C' have shares $[\![s]\!]^{C'}$, where $[\![s]\!]^{C}$ was the input provided to the protocol. Furthermore, privacy of s is satisfied regardless of whether the parties abort or not.*

If multiple shared elements $[\![s_1]\!]^{C}, \dots, [\![s_L]\!]^{C}$ are to be proactivized, we can make use of the fact that the signature scheme described in Section 5.4 allows for cheap signing and verification of long messages without penalty in communication. Further optimizations are presented in Appendix 5.7.

Communication Complexity. The communication complexity of the Π_{PSS} protocol is $\mathcal{C}_{\mathsf{PSS}}(L+1) + \mathcal{C}_{\mathsf{Sign}}(L) + \mathcal{C}_{\mathsf{Verify}}(L)$. Recall that $\mathcal{C}_{\mathsf{Sign}}(L) = \mathcal{C}_{\mathsf{Coin}}(1) + \mathcal{C}_{\mathsf{DotProduct}}(L)$, and $\mathcal{C}_{\mathsf{Verify}}(L) = \mathcal{C}_{\mathsf{Rand}}(1) + \mathcal{C}_{\mathsf{ScalarMult}}(1)$ For the case of Shamir secret sharing, $\mathcal{C}_{\mathsf{Rand}}(1) = 2n \log |\mathbb{F}|$, using the protocol from [58] and amortizing over multiple calls to $\mathcal{F}_{\mathsf{Rand}}$. Also, $\mathcal{C}_{\mathsf{DotProduct}}(L) = 5.5n \log |\mathbb{F}|$, and $\mathcal{C}_{\mathsf{ScalarMult}}(1) = 5.5n \log |\mathbb{F}|$ too, using the specialized bilinear protocol $\Pi^{\mathsf{shm}}_{\mathsf{DotProduct}}$ for Shamir SS described in Section 5.7. We ignore the cost $\mathcal{C}_{\mathsf{Coin}}(1)$ since it can be instantiated non-interactively using a PRG.

Given the above, the total communication complexity of the Π_{PSS} protocol is

$$\log(|\mathbb{F}|) \cdot ((L+1) \cdot n \cdot (n+1) + 13n) \text{ bits.}$$

Comparison with CHURP. The dynamic PSS protocol proposed in [113], is to our knowledge state-of-the-art in terms of communication complexity. At a high level, CHURP is made of two main protocols, Opt-CHURP, which is able to detect malicious behavior during the proactivization but is not able to point out which party or parties cheated, and Exp-CHURP, which performs

proactivization while enabling cheater detection at the expense of requiring more communication. Since in this work we have described a PSS protocol *with abort*, we compare our protocol against Opt-CHURP.

The total communication complexity of Opt-CHURP is $9Ln^2 \log |\mathbb{F}|$ bits in point-to-point channels, plus $256n$ bits over a blockchain,[5] so our novel method presents a 9-fold improvement over the state of the art. Furthermore, although not mentioned in our protocol, a lot of the communication that appears in the $13n$ term in our Π_{PSS} protocol can be regarded as preprocessing, that is, it is independent of the message being transmitted and can be computed in advance, before the proactivization phase.

Finally, we note that our novel protocol Π_{PSS} is conceptually much more simple than Opt-CHURP. Unlike in Opt-CHURP, our protocol does not require the expensive use of commitments and proofs at the individual level (i.e. *per party*) in order to ensure correctness of the transmitted value. Instead, we compute a *global* signature of the secret and check its validity after the proactivization.

5.6 Applications to Input Certification

MPC does not put any restriction on what kind of inputs are allowed, yet such a property has its place in many applications. For example, one might want to ensure that the two parties in the *millionaires problem* do not lie about their fortunes.

Signatures seem like the obvious candidate primitive for certifying inputs in MPC: A trusted party $\mathcal{T}$ will sign all inputs x_i of party P_i that need certification. Then, after P_i have shared its input $[\![x'_i]\!]$, which it may change if it is misbehaving, parties will verify that $[\![x'_i]\!]$ is a value that was previously signed by $\mathcal{T}$. While this approach clearly works (if P_i could get away with sharing x'_i, then P_i produced a forgery) it is nevertheless hindered by the fact that signature verification is expensive to compute on secret-shared values, arising from the fact that the usual first step in verifying a signature is hashing the message, which is prohibitely expensive in MPC. In this section we show that by using our secure PS signatures from Section 5.4, this approach is not longer infeasible, and in fact, it is quite efficient.

Certifying inputs with PS signatures

We consider a setting in which n parties $P_1, \dots, P_n$ wish to compute a function $f(\mathbf{x}_1, \dots, \mathbf{x}_n)$, where $\mathbf{x}_i \in \mathbb{F}^L$ corresponds to the input of party P_i. We assume that all parties hold the public key pk of some trusted authority $\mathcal{T}$, who provided each P_i with a PS signature (σ_1^i, σ_2^i) on its input $\mathbf{x}_i$. We also assume a functionality $\mathcal{F}_{\mathsf{Input}}$ that, on input $\mathbf{x}_i$ from P_i, distributes to the parties

[5] For a more detailed derivation of this complexity, see Section 5.7 in the appendix.

consistent shares $[\![x_{i1}]\!], \ldots, [\![x_{iL}]\!]$. We also assume the existence of a broadcast channel.

Our protocol, $\Pi_{\mathsf{CertInput}}$, allows a party P_i to distribute shares of its input, only if this input has been previously certified.

Protocol 8 Protocol $\Pi_{\mathsf{CertInput}}$

Input: Index $i \in \{1, \ldots, n\}$ and $\big((x_i)_{i=1}^L, \sigma_1, \sigma_2\big)$ from P_j.

Output: $([\![x_i]\!])_i$ where $\mathsf{Verify}(\mathsf{pk}, ([\![x_i]\!]_i), (\sigma_1, \sigma_2)) = 1$, or abort.

1. P_j calls $\mathcal{F}_{\mathsf{Input}}$ to distribute $(([\![x_i]\!])_i, [\![\sigma_2]\!]_{\mathbb{G}_1})$. Also, P_j broadcasts σ_1 to all parties.
2. Parties call $[\![r]\!]_{\mathbb{G}_T} \leftarrow \Pi_{\mathsf{Verify}}(\mathsf{pk}, ([\![x_i]\!])_{i=1}^L, \sigma_1, [\![\sigma_2]\!]_{\mathbb{G}_1})$.
3. Parties open $[\![r]\!]_{\mathbb{G}_T}$, who output $([\![x_i]\!])_i$ if $r = 1_{\mathbb{G}_T}$ and abort otherwise.

The security of the protocol follows seamlessly from the unforgeability of the PS signatures, proven in [126]. Optimizations are discussed in 5.7.

Complexity analysis. The communication complexity of the protocol $\Pi_{\mathsf{CertInput}}$ is $\mathcal{C}_{\mathsf{Input}}(L) + \mathcal{C}_{\mathsf{Verify}}(L) + \mathcal{C}_{\mathsf{Open}}(1)$ bits.

5.7 Implementation and Benchmarking

We implemented our protocols with the RELIC toolkit [10] using the pairing-friendly BLS12-381 curve. This curve has embedding degree $k = 12$ and a 255-bit prime-order subgroup, and became popular after it was adopted by the ZCash cryptocurrency [24]. It is now in the process of standardization due to its attractive performance characteristics, including an efficient towering of extensions, efficient GLV endomorphisms for scalar multiplications, cyclotomic squarings for fast exponentiation in $\mathbb{G}_T$, among others. In terms of security, the choice is motivated by recent attacks against the DLP in $\mathbb{G}_T$ [106] and are supported by the analysis in [115]. Our implementations make use of all optimizations implemented in RELIC, including Intel 64-bit Assembly acceleration, and extend the supported algorithms to allow computation of arbitrarily-sized linear combinations of $\mathbb{G}_2$ points through Pippenger's algorithm. We take special care to batch operations which can performed simultaneously, for example merging scalar multiplications together or combining the two pairing computations within MPC signature verification as a product of pairings. We deliberately enabled the variable-time but faster algorithms in the library relying on the timing-attack resistance built in MPC, since computations will

Operation		**Local (cc)**	**Two-party (cc)**
Scalar multiplication in $\mathbb{G}_1$		386	840
Scalar multiplication in $\mathbb{G}_2$		1,009	2,417
Exponentiation in $\mathbb{G}_T$		1,619	4,483
Pairing computation		3,107	4,063
PS key generation	(1 msg)	2,670	4,723
PS signature computation	(1 msg)	626	532
PS signature verification	(1 msg)	5,153	11,514
PS key generation	(10 msgs)	11,970	23,464
PS signature computation	(10 msgs)	656	532
PS signature verification	(10 msgs)	11,131	16,216

Table 5.1: Efficiency comparison between local computation and two-party computation of the main operations in pairing groups and PS signature computation/verification. We display execution times in 10^3 clock cycles (cc) for each of the main operations in the protocols and report the average for each of the two parties.

be performed essentially over ephemeral data. The resulting code will be contributed back to the library.

We benchmarked our implementation on an Intel Core i7-7820X Skylake CPU clocked at 3.6GHz with HyperThreading and TurboBoost turned off to reduce noise in the benchmarks. Each procedure was executed 10^4 times and the averages are reported in Table 5.1. It can be seen from the table that the MPC versions of scalar multiplications and exponentiations introduce a computational overhead ranging from 2.17 to 2.77, while pairing computation becomes only 30% slower. We notice that performance impact is higher for exponentiation in $\mathbb{G}_T$ due to a less efficient implementation in RELIC. This is justified by the lower prevalence of such operations in pairing-based protocols compared to operations in $\mathbb{G}_1$ and $\mathbb{G}_2$. For the PS protocol, key generation and signature verification in MPC are penalized in comparison to local computation approximately by a 2-factor, while the cost of signature computation stays essentially the same. There is no performance penalty for signature computation involving many messages because of the batching possibility in the PS signature scheme.

Certified Inputs

Here we compare our protocol for input certification from Section 5.6 with the experimental results reported in [26]. To perform a fair comparison, we converted the timings from the second half of Table 2 in [26] to clock cycles using the reported CPU frequency of 2.1GHz for an Intel Sandy Bridge Xeon E5-2620 machine. Each procedure in our implementation was executed 10^4 times for up to 10^2 messages, after which we decreased the number of executions linearly with

	Number of messages						
	1	10	10^2	10^3	10^4	10^5	10^6
Ours	11,514	16,216	62,714	357,445	2,334,742	22,281,049	220,572,619
[26]	11,445	18,690	103,950	970,200	9,723,000	111,090,000	-

Table 5.2: Efficiency comparison between our certified input protocol from Section 5.6 and the one presented in [26]. Numbers are measured in thousands of clock cycles (cc).

the increase in number of messages. We used as reference the largest running time of the two running parties (input provider and other party) reported in [26], since the computation would be bounded by the maximum running time. Our results are shown in Table 5.2, and show that our implementations are competitive for small numbers of messages, but improve on related work by a factor of 2–5 when the number of messages is at least 100. While the two benchmarking machines are different (Intel Sandy Bridge and Skylake), our implementations do not make use of any performance feature specific to Skylake, such as more advanced vector instruction sets. Hence we claim that the performance of our implementations would not be different enough in Sandy Bridge to explain the difference, and just converting performance figures to clock cycles makes the results generally comparable.

Appendix

Appendix to Chapter 5

Proof of Lemma 1

Proof. Suppose that $([\![x_i]\!]_{k+s}, [\![y_i]\!]_{k+s}, [\![z_i]\!]_{k+s})$ is the multiplication triple corresponding to the i-th multiplication gate, where $[\![x_i]\!]_{k+s}, [\![y_i]\!]_{k+s}$ are the sharings on the input wires and $[\![z_i]\!]_{k+s}$ is the sharing on the output wire. We note that the values on the input wires may not actually be the appropriate values as when the circuit is computed by honest parties. However, in the verification step, each gate is examined separately, and all that is important is whether the randomized result is $[\![r \cdot z_i]\!]_{k+s}$ for whatever z_i is here (i.e., even if an error was added by the adversary in previous gates). By the definition of $\mathcal{F}_{\mathsf{Mult}}$, a malicious adversary is able to carry out an additive attack, meaning that it can add a value to the output of each multiplication gate. We denote by $\delta_i \in \mathbb{Z}_{2^{k+s}}$ the value that is added by the adversary when $\mathcal{F}_{\mathsf{Mult}}$ is called with $[\![x_i]\!]_{k+s}$ and $[\![y_i]\!]_{k+s}$, and by $\gamma_i \in \mathbb{Z}_{2^{k+s}}$ the value added by the adversary when $\mathcal{F}_{\mathsf{Mult}}$ is called with the shares $[\![y_i]\!]_{k+s}$ and $[\![r \cdot x_i]\!]_{k+s}$. However, it is possible that the adversary has attacked previous gates and so $[\![y_i]\!]_{k+s}$ is actually multiplied with $[\![r \cdot x_i + \epsilon_i]\!]$, where the value $\epsilon_i \in \mathbb{Z}_{2^{k+s}}$ is an accumulated error from previous gates.[6] Thus, it holds that $\mathsf{val}([\![z_i]\!])^H = x_i \cdot y_i + \delta_i$ and $\mathsf{val}([\![r \cdot z_i]\!])^H = (r \cdot x_i + \epsilon_i) \cdot y_i + \gamma_i$. Similarly, for each input wire with sharing $[\![v_m]\!]$, it holds that $\mathsf{val}([\![r \cdot v_m]\!])^H = r \cdot v_m + \xi_m$, where $\xi_m \in \mathbb{Z}_{2^{k+s}}$ is the value added by the adversary when $\mathcal{F}_{\mathsf{Mult}}$ is called with $[\![r]\!]_{k+s}$ and the shared input

[6] Although attacks in previous gates may be carried out on both multiplications, the idea is here is to fix x_i which is shared by $[\![x_i]\!]_{k+s}$ at the current value on the wire, and then given the randomized sharing $[\![x'_i]\!]_{k+s}$, define $\epsilon_i = x'_i - r \cdot x_i$ as the accumulated error on the input wire.

$[\![v_m]\!]_{k+s}$. Thus, we have that

$$\begin{aligned}\mathsf{val}([\![u]\!])^H &= \sum_{i=1}^{N} \alpha_i \cdot ((r \cdot x_i + \epsilon_i) \cdot y_i + \gamma_i) \\ &\qquad + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m + \xi_m) + \Theta_1 \\ \mathsf{val}([\![w]\!])^H &= \sum_{i=1}^{N} \alpha_i \cdot (x_i \cdot y_i + \delta_i) + \sum_{m=1}^{M} \beta_m \cdot v_m + \Theta_2\end{aligned}$$

where $\Theta_1 \in \mathbb{Z}_{2^{k+s}}$ and $\Theta_2 \in \mathbb{Z}_{2^{k+s}}$ are the values being added by the adversary when $\mathcal{F}_{\mathsf{DotProduct}}$ is called in the verification step, and so

$$\begin{aligned}\mathsf{val}([\![T]\!])^H &= \mathsf{val}([\![u]\!])^H - r \cdot \mathsf{val}([\![w]\!])^H = \\ &= \sum_{i=1}^{N} \alpha_i \cdot ((r \cdot x_i + \epsilon_i) \cdot y_i + \gamma_i) + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m + \xi_m) + \theta_1 \\ &\qquad - r \cdot \left(\sum_{i=1}^{N} \alpha_i \cdot (x_i \cdot y_i + \delta_i) + \sum_{m=1}^{M} \beta_m \cdot v_m + \Theta_2 \right) \\ &= \sum_{i=1}^{N} \alpha_i \cdot (\epsilon_i \cdot y_i + \gamma_i - r \cdot \delta_i) \qquad (1) \\ &\qquad + \sum_{m=1}^{M} \beta_m \cdot \xi_m + (\Theta_1 - r \cdot \Theta_2),\end{aligned}$$

where the second equality holds because r is opened and so the multiplication $r \cdot [\![w]\!]_{k+s}$ always yields $[\![r \cdot w]\!]_{k+s}$. Let $\Delta_i = \epsilon_i \cdot y_i + \gamma_i - r \cdot \delta_i$.

Our goal is to show that $\mathsf{val}([\![T]\!])^H$, as shown in Eq. (2), equals 0 with probability at most $2^{-s+\log(s+1)}$. We have the following cases.

- *Case 1: There exists $m \in [M]$ such that $\xi_m \not\equiv_k 0$.* Let m_0 be the smallest such m for which this holds. Then $\mathsf{val}([\![T]\!])^H \equiv_{k+s} 0$ if and only if

$$\beta_{m_0} \cdot \xi_{m_0} \equiv_{k+s} \left(-\sum_{i=1}^{N} \alpha_i \cdot \Delta_i - \sum_{\substack{m=1 \\ m \neq m_0}}^{M} \beta_m \cdot \xi_m - (\Theta_1 - r \cdot \Theta_2) \right).$$

Let 2^u be the largest power of 2 dividing ξ_{m_0}. Then we have that

$$\beta_{m_0} \equiv_{k+s-u} \left(\frac{-\sum_{i=1}^{N} \alpha_i \cdot \Delta_i - \sum_{\substack{m=1 \\ m \neq m_0}}^{M} \beta_m \cdot \xi_m - (\Theta_1 - r \cdot \Theta_2)}{2^u} \right) \cdot \left(\frac{\xi_{m_0}}{2^u} \right)^{-1}.$$

By the assumption that $\xi_m \not\equiv_k 0$ it follows that $u < k$ and so $k + s - u > s$ which means that the above holds with probability at most 2^{-s}, since β_{m_0} is uniformly distributed over $\mathbb{Z}_{2^{k+s}}$.

- *Case 2: All* $\xi_m \equiv_k 0$. By the assumption in the lemma, some additive value $d \not\equiv_k 0$ was sent to $\mathcal{F}_{\mathsf{Mult}}$. Since none was sent for the input randomization, there exists some $i \in \{1, \ldots, N\}$ such that $\delta_i \not\equiv_k 0$ or $\gamma_i \not\equiv_k 0$. Let i_0 be the smallest such i for which this holds. Note that since this is the first error added which is $\not\equiv_k 0$, it holds that $\epsilon_{i_0} \equiv_k 0$. Thus, in this case, $\mathsf{val}([\![T]\!])^H \equiv_{k+s} 0$ if and only if $\alpha_{i_0} \cdot \Delta_{i_0} \equiv_{k+s} Y$, where

$$Y = \left(- \sum_{\substack{i=1 \\ i \neq i_0}}^{N} \alpha_i \cdot \Delta_i - \sum_{m=1}^{M} \beta_m \cdot \xi_m - (\Theta_1 - r \cdot \Theta_2) \right).$$

  Let q be the random variable corresponding to the largest power of 2 dividing Δ_{i_0}, where we define $q = k + s$ in the case that $\Delta_{i_0} \equiv_{k+s} 0$. Let E denote the event $\alpha_{i_0} \cdot \Delta_{i_0} \equiv_{k+s} Y$. We have the following claims.

  - *Claim 1: For $k < j \leq k + s$, it holds that* $\Pr[q = j] \leq 2^{-(j-k)}$.

    To see this, suppose that $q = j$ and $j > k$. It holds then that $\Delta_{i_0} \equiv_j 0$, and so $\Delta_{i_0} \equiv_k 0$. We first claim that in this case it must hold that $\delta_{i_0} \not\equiv_k 0$. Assume in contradiction that $\delta_{i_0} \equiv_k 0$. In addition, by our assumption we have that $\gamma_{i_0} \not\equiv_k 0$, $\epsilon_i \equiv_k 0$ and $\Delta_{i_0} = \epsilon_{i_0} \cdot y_{i_0} + \gamma_{i_0} - r \cdot \delta_{i_0} \equiv_k 0$. However, $\epsilon_i \cdot y_{i_0} \equiv_k 0$ and $r \cdot \delta_{i_0} \equiv_k 0$ imply that $\gamma_{i_0} \equiv_k 0$, which is a contradiction.

    We thus assume that $\delta_{i_0} \not\equiv_k 0$, and in particular there exists $u < k$, such that u is the largest power of 2 dividing δ_{i_0}. It is easy to see then that $q = j$ implies that $r \equiv_{j-u} \left(\frac{\epsilon_{i_0} \cdot y_{i_0} + \gamma_{i_0}}{2^u}\right) \cdot \left(\frac{\delta_{i_0}}{2^u}\right)^{-1}$. Since $r \in \mathbb{Z}_{2^{k+s}}$ is uniformly random and $u < k$, we have that this equation holds with probability of at most $2^{-(j-u)} \leq 2^{-(j-k)}$.

  - *Claim 2: For $k < j < k + s$ it holds that* $\Pr[E \mid q = j] \leq 2^{-(k+s-j)}$.

    To prove this let us assume that $q = j$ and that E holds. In this case we can write $\alpha_{i_0} \equiv_{k+s-j} \frac{Y}{2^j} \cdot \left(\frac{\Delta_{i_0}}{2^j}\right)^{-1}$. For $k < j < k + s$ it holds that $0 < k + s - j < s$ and therefore this equation can be only satisfied with probability at most $2^{-(k+s-j)}$, given that $\alpha_{i_0} \in \mathbb{Z}_{2^s}$ is uniformly random.

  - *Claim 3:* $\Pr[E \mid 0 \leq q \leq k] \leq 2^{-s}$.

    This is implied by the proof of the previous claim, since in the case that $q = j$ with $0 \leq j \leq k$, it holds that $k + s - j \geq s$, so the event E implies that $\alpha_{i_0} \equiv_s \frac{Y}{2^j} \cdot \left(\frac{\Delta_{i_0}}{2^j}\right)^{-1}$, which holds with probability at most 2^{-s}.

Putting these pieces together, we thus have the following:

$$\begin{aligned}\Pr[E] &= \Pr[E \mid 0 \leq q \leq k] \cdot \Pr[0 \leq q \leq k] + \\ &\quad \sum_{j=k+1}^{k+s} \Pr[E \mid q = j] \cdot \Pr[q = j] \\ &\leq 2^{-s} + s \cdot 2^{-s} = (s+1) \cdot 2^{-s} = 2^{-s+\log(s+1)}. \end{aligned} \tag{2}$$

To sum up the proof, in the first case we obtained that $T = 0$ with probability of at most 2^{-s} whereas in the second case, this holds with probability of at most $2^{-s+\log(s+1)}$. Therefore, we conclude that the probability that $T = 0$ in the verification step is bounded by $2^{-s+\log(s+1)}$ as stated in the lemma. This concludes the proof. □

Instantiation for 3 parties

We now present in detail the efficient three party instantiation of our compiler from replicated secret sharing. Sharing a value $x \in \mathbb{Z}_{2^\ell}$ is done by picking at random $x_1, x_2, x_3 \in \mathbb{Z}_{2^\ell}$ such that $\sum_i x_i \equiv_\ell x$. P_i's share of x is the pair (x_i, x_{i+1}) and we use the convention that $i + 1 = 1$ when $i = 3$. To reconstruct a secret, P_i receives the missing share from the two other parties. Note that reconstructing a secret is robust in the sense that parties either reconstruct the correct value x or they abort.

Replicated secret sharing satisfies the properties described in Section 3.2, and one can efficiently realize the required functionalities described in the same section. Below we discuss some of these properties/functionalities.

Generating Random Shares

Shares of a random value can be generated non-interactively, as noted in [111, 117], by making use of a setup phase in which each party P_i obtains shares of two random keys k_i, k_{i+1} for a pseudorandom function (PRF) F. The parties generate shares of a random value for the j-th time by letting P_i's share to be (r_i, r_{i+1}), where $r_i = F_{k_i}(j)$. These are replicated shares of the (pseudo)random value $r = \sum_i F_{k_i}(j)$. Proving that this securely computes $\mathcal{F}_{\mathsf{Rand}}$ is straight forward and we omit the details.

Secure Multiplication up to an Additive Attack

To multiply two shared values, we use the protocol from [8, 117], which is described in 9. The shares of 0 that this protocol needs can be obtained by using correlated keys for a PRF, in similar fashion to the protocol for $\mathcal{F}_{\mathsf{Rand}}$ sketched above.

Protocol 9 Secure multiplication up to an additive attack.

Inputs: Parties hold sharings $[\![x]\!]$, $[\![y]\!]$ and additive sharings $(\alpha_1, \alpha_2, \alpha_3)$ where $\sum_{i=1}^{3} \alpha_i = 0$.

1. P_i computes $z_i = x_i y_i + x_{i+1} y_i + x_i y_{i+1} + \alpha_i$ and sends z_i to P_{i-1}.
2. P_j, upon receiving z_{j+1}, defines its share of $[\![x \cdot y]\!]$ as (z_j, z_{j+1}).

The above protocol is secure up to an additive attack as noted in [111]. We note that this can be extended to instantiate $\mathcal{F}_{\mathsf{DotProduct}}$ at the communication cost of one single multiplication, as shown in [45].

Efficient Checking Equality to 0

Checking that a value is a share of 0 can be performed very efficiently in this setting by relying on a Random Oracle $\mathcal{H}$. The observation we rely on is that, if $\sum_i x_i \equiv_\ell 0$, then $x_{i-1} \equiv_\ell -(x_i + x_{i+1})$ and so P_i can send $z_i = \mathcal{H}(-(x_i + x_{i+1}))$ which will be equal to x_{i-1} which is held by P_{i+1} and P_{i-1}. Since only one party is corrupted, it suffices that each P_i will send it only to P_{i+1}. Upon receiving z_i from P_i, P_{i+1} checks that $z_i = \mathcal{H}(x_{i-1})$ and aborts if this is not the case.

This protocol is formalized in Protocol 10 in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model. The $\mathcal{F}_{\mathsf{RO}}$ functionality is described in Functionality 1. We remark that that this protocol does not instantiate $\mathcal{F}_{\mathsf{CheckZero}}$ exactly. In order for the proof of security to work, we need to allow the adversary to cause the parties to reject also when $v = 0$. We denote this modified functionality by $\mathcal{F}'_{\mathsf{CheckZero}}$. This is minor change since the main requirement from $\mathcal{F}_{\mathsf{CheckZero}}$ in our compiler is that the parties won't accept a value as 0 when it is not, which is still satisfied by the modified functionality.

Functionality 1 $\mathcal{F}_{\mathsf{RO}}$ – Random Oracle functionality

---

Let M be an initially empty map.

1. On input x from a party P, if $(x, y) \in M$ for some y, return y. Otherwise pick y at random and set $M = \{(x, y)\} \cup M$ and return y.
2. On (x, y) from § and if $(x, \cdot) \notin M$ set $M = \{(x, y)\} \cup M$.

Protocol 10 Checking Equality to 0 in the $\mathcal{F}_{\mathsf{RO}}$-Hybrid Model

Inputs: Parties hold a sharing $[\![v]\!]$.

1. Party P_i queries $\beta_i \leftarrow \mathcal{F}_{\mathsf{RO}}(-(v_i + v_{i+1}))$ and sends β_i to P_{i+1}.
2. Upon receiving β_{i-1} from P_{i-1}, each party P_i checks that $\beta_{i-1} = \mathcal{F}_{\mathsf{RO}}(v_{i+1})$. If this is not the case, then P_i outputs `reject`. Otherwise, it outputs `accept`.

We have the following proposition.

Proposition 3. *Protocol 10 securely computes $\mathcal{F}_{\mathsf{CheckZero}}$ in the $\mathcal{F}_{\mathsf{RO}}$-hybrid model in the presence of one malicious corrupted party.*

Proof. Let $\mathcal{A}$ be the real adversary who corrupts at most one party and § the ideal world simulator. Let P_i be the corrupted party. The simulation begins with § receiving the shares of P_i, i.e., (v_i, v_{i+1}). Then, § proceed as follows:

- If § receives `accept` from $\mathcal{F}'_{\mathsf{CheckZero}}$, then it knows that $v \equiv_\ell 0$ and so it can compute the share $v_{i-1} = -(v_i + v_{i+1})$ and so it knows the honest parties' shares and can perfectly simulate the execution, while playing the role of $\mathcal{F}_{\mathsf{RO}}$. If $\mathcal{A}$ cause the parites to reject by using different shares, then § sends `reject` to $\mathcal{F}'_{\mathsf{CheckZero}}$.
- If § receives `reject`, then it chooses a random $v_{i-1} \in \mathbb{Z}_{2^\ell} \setminus \{-(v_i + v_{i+1})\}$ and defines the honest patries' shares accordingly. Then, it plays the role of $\mathcal{F}_{\mathsf{RO}}$ simulating the remaining of the protocol. By the definition of $\mathcal{F}_{\mathsf{RO}}$, the view of $\mathcal{A}$ is distributed identically in the simulated and the real execution.

□

Appendix to Section 3.5 - Shamir-SS Instantiation

Proof of Lemma 2: Securely Computing $\mathcal{F}_{\mathsf{Rand}}$

Lemma 7 (Lemma 2 - restated)**.** *Protocol 3 securely computes $(n-\tau)d$ parallel invocations of $\mathcal{F}_{\mathsf{Rand}}$ for $[\![\cdot]\!]_\tau$ with statistical error of at most $2^{-\kappa}$ in the presence of a malicious adversary controlling $t < n/2$ parties.*

Proof. Let $\mathcal{A}$ be the real-world adversary. The simulator $\mathcal{S}$ interacts with $\mathcal{A}$ by simulating the honest parties in an execution of the protocol. In doing so, $\mathcal{S}$ obtains honest parties' shares $\langle r_1 \rangle_H, \ldots, \langle r_{n-\tau} \rangle_H$.

We distinguish three cases:

1. If at least one of the simulated honest parties aborts in any of the executions of Protocol 4, then $\mathcal{S}$ sends `abort` to $\mathcal{F}_{\mathsf{Rand}}$.

2. If the checks pass but the honest parties' shares are inconsistent, $\mathcal{S}$ outputs `fail`. By Lemma 3 this only happens with probability at most $2^{-\kappa}$, allowed by the claim.

3. In the remaining case, the checks of Protocol 4 pass and the honest parties' shares are consistent. $\mathcal{S}$ calculates the corrupted parties' shares $\langle r_1 \rangle_C, \ldots, \langle r_{n-\tau} \rangle_C$ from the honest parties' shares, and sends them to $\mathcal{F}_{\mathsf{Rand}}$.

Before the invocation of $\mathcal{F}_{\mathsf{Rand}}$, the honest parties have no private inputs, hence $\mathcal{S}$ simulates them perfectly and $\mathcal{A}$'s view will be identical to the real execution. Thus, the simulated honest parties will abort in the ideal execution precisely when they would in the real execution.

The only thing it remains to prove is that if the parties do not abort, the output shares are identically distributed in the real and ideal executions. In particular, we need to prove that in the real execution, the *sharings* are independent and uniformly sampled from $\langle \cdot \rangle$.

Let $H' \subseteq H$ be a subset of honest parties of size $n - \tau$, and let $C := \{1, \ldots, n\} \setminus H$ denote its complement. Let A_H, A_C denote the submatrices of A corresponding to the columns indexed by H' and C respectively. Let $\langle \backslash_H \rangle$ denote the vector $\langle s_i \rangle_{i \in H}$ of length $n - \tau$, and correspondingly $\langle \backslash_C \rangle := \langle \backslash_i \rangle_{i \in C}$. Then $(\langle r_1 \rangle, \ldots, \langle r_{n-\tau} \rangle)^T = A_H \langle \backslash_H \rangle + A_C \langle \backslash_C \rangle$. Since $\langle \backslash_{H'} \rangle$ is wholly generated by the honest parties, it consists of $n - \tau$ independent and uniformly random sharings of $\langle \cdot \rangle$. A_H is invertible (since A is hyperinvertible), hence we also have that $\langle \backslash_{H'} \rangle$ consists of independent and uniformly random sharings. Adding a fixed $A_C \langle \backslash_C \rangle$ will not affect the distribution, hence the sharings $\langle r_1 \rangle, \ldots, \langle r_{n-\tau} \rangle$ are independent and uniformly random sharings. □

Proof of Lemma 5: Securely Computing $\mathcal{F}_{\mathsf{Mult}}$

Lemma 8 (Lemma 5 - restated). *Protocol 5 securely computes $\mathcal{F}_{\mathsf{Mult}}$ with statistical error $\leq 2^{-\kappa}$ in the $\mathcal{F}_{\mathsf{Rand}}$-hybrid model in the presence of a malicious adversary controlling $t < n/2$ parties.*

Proof. Without loss of generality, assume $2\tau + 1 = n$ (recall that τ is the secret sharing threshold and not the number of corrupted parties, and so the proof still holds for any $t < n/2$).

For the offline phase, the simulator acts as in Lemma 2. By the proof, we have that $[\![r]\!]_\tau$ is a correct sharing. The sharing $[\![r']\!]_{(2\tau)}$ is not well-defined, because the adversary can change its mind about its shares at any time. However, the adversary always knows the additive error $r' - r$ that it introduces by changing its shares.

For the online phase, $\mathcal{S}$ simulates the honest parties towards $\mathcal{A}$.
We distinguish two cases:

- *Case 1: P_1 is not corrupt.* The simulated P_1 receives $\{u_i\}_{i\in C}$ from $\mathcal{A}$. If it receives $\perp$ for any value u_i, it sends `abort` to $\mathcal{F}_{\mathsf{Mult}}$ and simulates P_1 aborting. Otherwise, it calls $\mathcal{F}_{\mathsf{Mult}}$ and receives $\{x_i\}_{i\in C}, \{y_i\}_{i\in C}$. For any $i \in C$, since $\mathcal{S}$ knows x_i, y_i, r'_i, it may calculate $\delta_i = x_i y_i - r'_i$ and thus the value $\pi(\lambda_i \delta_i)$ the adversary is supposed to send if it behaves honestly. The simulator can therefore extract $d = \sum_{i\in C} u_i - \pi(\lambda_i\delta_i)$. $\mathcal{S}$ does not know the true value of δ, however it may sample $\delta \leftarrow \mathbb{Z}_{2^\ell}$, send it to the corrupted parties, and calculate the corrupted parties' shares as $z_i = r_i + \delta + d$.

  It then simulates the broadcast of δ. If the broadcast aborts, $\mathcal{S}$ simulates the parties aborting and sends `abort` to $\mathcal{F}_{\mathsf{Mult}}$. Otherwise, it sends $d, \{z_i\}_{i\in C}$ to $\mathcal{F}_{\mathsf{Mult}}$, and outputs whatever $\mathcal{A}$ outputs.

  In the ideal execution, $\mathcal{A}$ receives a random δ. It cannot distinguish this from the real value $x \cdot y - r$, since r is uniformly random and by privacy of the secret-sharing scheme it does not have any information on it.

- *Case 2: P_1 is corrupt.* $\mathcal{S}$ samples $[\![\delta]\!]_{(2\tau)} \leftarrow [\![\cdot]\!]_{(2\tau)}$. For $i \in H$ it sends $u_i = \pi(\lambda_i\delta_i)$ to the corrupted P_1. The simulated honest parties receive an identical broadcasted value δ', otherwise the broadcast protocol aborts.

  Since $\mathcal{S}$ knows δ, it can extract $d := \delta' - \delta$, and calculate the corrupted parties' shares as $z_i = r_i + \delta'$. It then sends $d, \{z_i\}_{i\in C}$ to $\mathcal{F}_{\mathsf{Mult}}$, and it outputs whatever $\mathcal{A}$ outputs.

As mentioned above, the adversary cannot distinguish whether it is talking to a simulator or the real parties, hence its output will be identical.

In the ideal execution where no abort took place, the actual (non-simulated) parties receive their shares $\{z_i\}_{i\in H}$ directly from $\mathcal{F}_{\mathsf{Mult}}$. The shares are consistent and will reconstruct to the secret $z = x \cdot y + d$. In the ideal execution, the shares are generated by the probabilistic function $\mathsf{share}(z, \{z_i\}_{z\in C})$, such that the shares are uniformly random subject to the constraints on the shares.[7] In the real execution, the shares also correspond to z. The sharing in the real execution is calculated as $[\![r]\!]_\tau + \delta$, where $[\![r]\!]_\tau$ is a uniformly random sharing. Therefore, the outputs are identical in both executions. □

Reducing Communication Using Pseudo-Randomness [30, 119]

Our protocol as described so far is information-theoretic. We can reduce communication by using a pseudo-random generator in the following way. Assume that each pair of parties hold a joint random seed. Then, when party

[7] Depending on the privacy threshold the constraints may fully determine the shares.

P_i shares an element with degree t, it is possible to derive t shares from the seed known to P_i and the corresponding party, and set the remaining $t+1$ shares (including the dealer's own share) given the pseudo-random shares and the value of the secret. Thus, only t shares need to be transmitted, thereby reducing communication by half. Using the same reasoning, it is possible to share a secret using $2t$-degree without any interaction. Here $n-1=2t$ shares are computed using the seed known to the dealer and each party, and then the dealer sets its own share such that all shares will reconstruct to the secret. We can use this idea to also reduce communication in the multiplication protocol. Instead of broadcasting δ, party P_i can share it to the parties with degree t, and use the above optimization, such that P_1 will have to send t elements instead of $n-1$. We note that here instead of comparing δ (to ensure correctness of output sharings), the parties can perform a batch correctness check (Protocol 4) for all sharings dealt by P_1 before the verification step in the main protocol.

Appendix to Chapter 6

Related work on secure inference

Secure evaluation of Neural Networks can be traced back to at least the work by Orlandi et al. [16, 120] which present a solution based on HE techniques. Several later works rely on HE techniques either in full or in part. CryptoNets [80] use Leveled Homomorphic Encryption (LHE), which necessitates bounding the number of operations a priori. In addition, HE only permits evaluation of polynomials and as such cannot compute e.g., the Rectified Linear activation functions (the function $x \mapsto \max(0, x)$) and the authors therefor rely on the approximation $x \mapsto x^2$. However, and as pointed out by Gilad-Bachrach et al. [80], such an approximation makes training difficult for larger networks, the issue being that the derivative of x^2 is unbounded. Chabanne et al. [40] improve upon CryptoNets by evaluating networks with 6 hidden layers (as opposed to only 2 as in Gilad-Bachrach et al.). More recently, Bourse et al. [31] obtain faster evaluation albeit for a smaller network (one and two hidden layers) by combining FHE and Discretized Neural Networks (i.e., networks where weights are in $\{1, -1\}$).

One of the downsides of HE based solutions are their inefficiency and inability to handle common activation functions. Gazelle [100] combines garbled circuits (GC) with additive HE (AHE) in order to obtain a more efficient system. The boost in efficiency is attributed to an efficient method of switching between the AHE scheme and a GC, where the former is used to compute convolutions and fully connected layers, while the latter is used to compute the network's activation functions.

The idea of using multiple different protocols to achieve faster predictions have been used before [100]. MiniONN [112] develops a technique for turning a pretrained model into an oblivious one, which can be evaluated using a mix of HE, additive secret sharing and GC. Chameleon [130], which is an extension of the ABY framework by Demmler et al. [65], likewise use secret sharing for matrix operations and GC for activation functions. More recently, ABY3 by Mohassel and Rindal [117], also benchmark secure evaluation (albeit the authors do not implement full inference) in a framework that relies on a mix of secret sharing, boolean (i.e., GMW) and garbled circuits.

Finally, like solutions relying purely on HE have been considered before,

so has solutions that rely purely on GC or MPC; the latter of which is most relevant to this work. DeepSecure [132] is perhaps the first work to take a pure GC based approach for evaluating Neural Networks. More recently XONN [131] builds a very efficient GC based solution by noting that Binarized Neural Networks [96] (i.e., networks with weights that are bits) can be evaluated very efficiently. XONN shows that evaluating deep networks (> 20 layers) is possible. A different approach is taken by Ball et al. [13] where the authors use the arithmetic garbling technique of Ball et al. [12] to evaluate Neural Networks. Pure MPC based solutions have been studied in SecureML [118], which employs a three-party honest majority protocol. A major performance boost in SecureML can be attributed to the way fixed point arithmetic is handled, where the authors show that it is possible to just have parties perform the truncation locally. SecureNN [142] can be seen as an extension of SecureML where both three- and four-party protocols (both with one corrupted party) are used. Concurrently to this work, CrypTFlow [109] builds a system on top of SecureNN that is capable of evaluating very large networks (>100 layers) in reasonable time. Another very attractive feature of CrypTFlow is that it provides a more complete framework that accepts standard Tensorflow trained models as input (hence the name).

MPC Protocols

Dishonest Majority

Protocols in the dishonest majority setting are often harder to develop and they are also more complex than honest majority ones. They are typically based in additive secret sharing and use authentication tags for active security to ensure that the openings of shared values are done correctly.
• SPDZ2k: This is the first actively secure protocol over $\mathbb{Z}_{2^k}$ in the dishonest majority setting, and it was proposed initially by Cramer et al. [51] and implemented subsequently by Damgård et al. [62]. This protocol can be seen as an extension of MASCOT [104] (itself being an extension of SPDZ, hence the name). Multiplications in SPDZ2k are handled using multiplication triples, which are preprocessed using oblivious transfer like in MASCOT. Authentication is handled like in SPDZ, but with an addition that allows this method to work over $\mathbb{Z}_{2^k}$ which consists of working over the ring $\mathbb{Z}_{2^{k+s}}$ and using the upper s bits for authentication.
• OTSemi2k, OTSemiPrime: These protocols denote cut-down versions of SPDZ2k and MASCOT, respectively. In particular, they omit the usage of authentication tags and the so-called “sacrifice” where two triples are checked against each other and only of them can subsequently used in the protocol. There essentially remains the generation of multiplication triples using OT.
• LowGear: This is an actively secure protocol for computation modulo a prime. It uses semi-homomorphic encryption based on learning with errors. See Keller

et al. [105] for details.

Honest Majority

Honest majority protocols are typically developed using Shamir Secret Sharing (for an arbitrary number of parties) or Replicated Secret Sharing (for small number of parties). Since we consider only a small number of servers we focus on the replicated SS instantiations.
• Replicated2k, ReplicatedPrime: This protocol secret-shares a value x among three parties P_1, P_2, P_3 by letting each P_i have random pairs (x_i, x_{i+1}) (indexes wrap around modulo 3) subject to $x \equiv x_1 + x_2 + x_3 \bmod M$, where $M = 2^k$ for the ring case and $M = p$ for the field case. The most efficient passively secure multiplication protocol to date is the one presented by Araki et al. [8], where the total communication involves 3 ring elements.
• PsReplicatedPrime: This protocol by Lindell and Nof [111] extends ReplicatedPrime to active security by preprocessing potentially incorrect triples and proceeding to the online phase using these, optimistically, checking their correctness at the end of the execution using sacrificing techniques.
• PsReplicated2k: This protocol by Eerikson et al. [67] is an extension of the one by Lindell et al. [111] to the ring setting. This is achieved by incorporating ideas by Cramer et al. [51] in order to adapt the post-sacrifice step by Lindell et al. to the ring $\mathbb{Z}_{2^k}$.

Extended results

Communication and preprocessing. Table 1 and Table 2, analogous to Table 4.3 and Table 4.4, presents the communication, in Gigabyes, used by the protocols we consider when evaluating different ImageNet models. As noted in Section 5.7, dishonest majority protocols require a great deal of preprocessing material in order to evaluate a network, which can be seen by the large differences between the values in columns corresponding to dishonest majority, with respect to honest majority. Interestingly, the protocol over $\mathbb{F}_p$ are cheaper with active security than the protocol over $\mathbb{Z}_{2^k}$. This is likely due to the fact that proprocessing in $\mathbb{F}_p$ (with active security) is more communication efficient, than the protocol over $\mathbb{Z}_{2^k}$, as illustrated in Table 1 and Table 2.

WAN Benchmarks. We have also run the smallest model in a WAN setting where each party is located on a different continent. For computation over rings with probablistic truncation, the timings range from 110 seconds for passive honest-majority computation to 28,000 seconds for active dishonest-majority computation.

Variant	Accuracy		Trunc.	Passive Security			
				Dishonest Maj.		Honest Maj.	
	Top-1	Top-5		$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$
V1 0.25_128	39.5%	64.4%	Prob.	199.2	37.1	0.1	3.4
			Exact	296.3	44.6	1.0	4.0
V1 0.25_160	42.8%	68.1%	Prob.	311.2	58.0	0.1	5.4
			Exact	462.8	69.6	1.5	6.2
V1 0.25_192	45.7%	70.8%	Prob.	447.5	83.4	0.1	7.7
			Exact	665.6	100.2	2.2	8.9
V1 0.25_224	48.2%	72.8%	Prob.	608.5	113.3	0.2	10.5
			Exact	905.3	136.3	3.0	12.2
V1 0.5_128	54.9%	78.1%	Prob.	438.0	79.1	0.1	6.9
			Exact	631.8	94.0	1.9	7.9
V1 0.5_160	57.2%	80.5%	Prob.	684.6	123.4	0.2	10.7
			Exact	987.4	146.8	3.0	12.4
V1 0.5_192	59.9%	82.1%	Prob.	984.8	177.6	0.3	15.5
			Exact	1420.7	211.3	4.4	17.9
V1 0.5_224	61.2%	83.2%	Prob.	1339.4	241.6	0.3	21.0
			Exact	1932.6	287.4	5.9	24.3
V1 0.75_128	55.9%	79.1%	Prob.	716.9	125.9	0.2	10.3
			Exact	1007.6	148.3	2.9	11.9
V1 0.75_160	62.4%	83.7%	Prob.	1120.8	196.7	0.3	16.1
			Exact	1574.8	231.7	4.6	18.6
V1 0.75_192	66.1%	86.2%	Prob.	1612.4	283.0	0.4	23.2
			Exact	2266.2	333.5	6.6	26.8
V1 0.75_224	66.9%	86.9%	Prob.	2193.2	385.0	0.5	31.5
			Exact	3082.9	453.7	8.9	36.5
V1 1.0_128	63.3%	84.1%	Prob.	1035.9	177.6	0.2	13.7
			Exact	1423.5	207.6	3.9	15.9
V1 1.0_160	66.9%	86.7%	Prob.	1619.6	277.6	0.4	21.5
			Exact	2224.9	324.4	6.1	24.8
V1 1.0_192	69.1%	88.1%	Prob.	2330.3	399.4	0.5	30.9
			Exact	3201.9	466.8	8.7	35.7
V1 1.0_224	70.0%	89.0%	Prob.	3169.9	543.5	0.7	42.0
			Exact	4356.2	635.1	11.9	48.6

Table 1: Communication complexity, in Gigabytes, of securely evaluating some of the networks in the MobileNets family with passive security, in a LAN network. The first number in variant is the width multiplier and the second is the resolution multiplier. Top-1 accuracy measures when the truth label is predicted correctly by the model whereas Top-5 measures when the truth label is among the first 5 outputs of the model.

Variant	Accuracy		Trunc.	Active Security			
				Dishonest Maj.		Honest Maj.	
	Top-1	Top-5		$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$
V1 0.25_128	39.5%	64.4%	Prob.	1748.4	282.4	2.5	8.7
			Exact	2578.2	335.3	4.7	10.3
V1 0.25_160	42.8%	68.1%	Prob.	2731.2	423.9	3.8	13.6
			Exact	4027.1	511.9	7.3	16.1
V1 0.25_192	45.7%	70.8%	Prob.	3927.0	600.0	5.4	19.6
			Exact	5792.2	723.3	10.4	23.2
V1 0.25_224	48.2%	72.8%	Prob.	5339.9	811.3	7.3	26.6
			Exact	7877.7	987.2	14.2	31.5
V1 0.5_128	54.9%	78.1%	Prob.	3834.5	581.7	5.8	18.5
			Exact	5492.6	687.4	10.2	21.7
V1 0.5_160	57.2%	80.5%	Prob.	5993.7	899.7	9.0	28.8
			Exact	8583.4	1075.6	15.9	33.8
V1 0.5_192	59.9%	82.1%	Prob.	8621.7	1286.9	12.9	41.5
			Exact	12349.8	1533.4	22.9	48.7
V1 0.5_224	61.2%	83.2%	Prob.	11725.6	1744.6	17.5	56.4
			Exact	16799.2	2079.1	31.2	66.3
V1 0.75_128	55.9%	79.1%	Prob.	6264.3	916.3	10.0	29.2
			Exact	8750.4	1074.9	16.7	34.1
V1 0.75_160	62.4%	83.7%	Prob.	9793.1	1428.3	15.6	45.7
			Exact	13676.4	1692.3	26.1	53.2
V1 0.75_192	66.1%	86.2%	Prob.	14089.1	2044.4	22.4	65.8
			Exact	19680.3	2432.0	37.5	76.6
V1 0.75_224	66.9%	86.9%	Prob.	19163.5	2783.4	30.5	89.5
			Exact	26773.0	3294.5	51.0	104.2
V1 1.0_128	63.3%	84.1%	Prob.	9037.9	1286.1	15.2	41.1
			Exact	12352.1	1514.8	24.2	47.5
V1 1.0_160	66.9%	86.7%	Prob.	14128.8	2009.9	23.7	64.2
			Exact	19306.2	2361.8	37.7	74.3
V1 1.0_192	69.1%	88.1%	Prob.	20328.5	2889.9	34.1	92.4
			Exact	27782.7	3400.7	54.2	106.9
V1 1.0_224	70.0%	89.0%	Prob.	27652.5	3928.3	46.4	125.8
			Exact	37798.3	4615.2	73.7	145.4

Table 2: As previous table, but active security.

Appendix to Chapter 7

Bilinear maps for MPC

We formalize the intuition from Section 5.2 below where we describe the protocol Π_{bilinear} in detail.

For this protocol we assume a functionality $\mathcal{F}_{\mathsf{OuterProd}}$ that produce random shares $[\![a_1]\!], \ldots, [\![a_d]\!], [\![b_1]\!], \ldots, [\![b_d]\!]$ over $\mathbb{F}$, together with $[\![a_i b_j]\!]$ for $i, j \in \{1, \ldots, d\}$. This is used to produce the "bilinear triples" mentioned earlier. (Notice further that the case where $d = 1$, $\mathcal{F}_{\mathsf{OuterProd}}$ corresponds to a classical triple-preprocessing functionality.) Also, in the protocol below we assume that $\{u_1, \ldots, u_d\}$ is a basis for U and that $\{v_1, \ldots, v_d\}$ is a basis for V.

Protocol 11 Protocol Π_{bilinear}

Inputs: $[\![u]\!]_U$ and $[\![v]\!]_V$.

Output: $[\![w]\!]_W$ where $w = \phi(u, v) \in W$.

Offline Phase:

1. The parties call $\left(\{[\![a_i]\!]\}_{i=1}^d, \{[\![b_i]\!]\}_{i=1}^d, \{[\![a_i b_j]\!]\}_{i,j=1}^d\right) \leftarrow \mathcal{F}_{\mathsf{OuterProd}}$.
2. The parties use the LSS homomorphisms $x \mapsto x \cdot u_i$ and $x \mapsto x \cdot v_i$ to locally compute $[\![\alpha]\!]_U = \sum_{i=1}^d [\![a_i]\!] \cdot u_i$ and $[\![\beta]\!]_V = \sum_{i=1}^d [\![b_i]\!] \cdot v_i$, respectively.
3. The parties compute $[\![\phi(a_i u_i, b_j v_j)]\!]_W \leftarrow [\![a_i b_j]\!] \cdot \phi(u_i, v_j)$ using the LSS homomorphisms $x \mapsto x \cdot \phi(u_i, v_j)$.
4. The parties compute locally $[\![\phi(\alpha, \beta)]\!]_W = \sum_{i,j=1}^d [\![\phi(a_i u_i, b_j v_j)]\!]_W$.

Online Phase:

1. The parties open $\delta \leftarrow [\![u]\!]_U - [\![\alpha]\!]_U$ and $\epsilon \leftarrow [\![v]\!]_V - [\![\beta]\!]_V$
2. The parties use the LSS homomorphism $\phi(\delta, \cdot)$ to compute $[\![\phi(\delta, \beta)]\!]_W \leftarrow \phi(\delta, [\![\beta]\!]_V)$, and similarly they use the LSS homomorphism $\phi(\cdot, \epsilon)$ to compute $[\![\phi(\alpha, \epsilon)]\!]_W \leftarrow \phi([\![\alpha]\!]_U, \epsilon)$.

---

3. The parties compute locally and output $[\![\phi(u,v)]\!]_W = \phi(\delta,\epsilon) + [\![\phi(\delta,\beta)]\!]_W + [\![\phi(\alpha,\epsilon)]\!]_W + [\![\phi(\alpha,\beta)]\!]_W$.

Proofs

Proof of Lemma 6

Proof. Note that $[\![\alpha]\!]_{\mathbb{G}_T}/[\![\beta]\!]_{\mathbb{G}_T} = [\![e(\sigma_1, X + \sum_i m_i Y_i)/e(\sigma_2, H)]\!]_{\mathbb{G}_T}$ which is $1_{\mathbb{G}_T}$ if and only if $e(\sigma_1, X + \sum_i m_i Y_i) = e(\sigma_2, H)$; that is, if the signature is valid. Thus we have that the distribution of $[\![b]\!]_{\mathbb{G}_T} = [\![(a/\beta)^\rho]\!]_{\mathbb{G}_T}$ is either uniformly random (if $\alpha \neq \beta$), or $1_{\mathbb{G}_T}$ (if $\alpha = \beta$). To see that $[\![b]\!]_{\mathbb{G}_T}$ is uniformly random when $\alpha \neq \beta$ it suffices to note that α/β is a generator of $\mathbb{G}_T$ and that ρ was picked at random. □

Proof of Theorem 2

Proof. We begin by introducing some notation. Let $A \subseteq C$ and $A' \subseteq C'$ be the corresponding subsets of corrupt parties. For an honest party P_i it should hold that $s_i = \sum_{j=1}^{t+1} s_{ij}$, where s_{ij} is the additive share sent by P_i to P_j in step 1. However, for $P_i \in \mathcal{A}$, this may not be the case, so we define $\delta_i \in \mathbb{F}$ such that $s_i + \delta_i = \sum_{j=1}^{t+1} s_{ij}$. Finally, each $P_i \in U$ is supposed to send a_{ij} in step 3, but naturally, parties in $A \cap U$ may not follow this. We define ϵ_{ij} for $P_i \in A \cap U$ and $j = 1, \ldots, n$ in such a way that $a_{ij} + \epsilon_{ij}$ is the value sent by P_i to P'_j in step 3.

It is easy to see that the value reconstructed by P'_j in step 4 is $s'_j = \sum_{i=1}^{t+1} a_{ij} = \epsilon_j + \delta_j + s_j + \sum_{k=0}^{t} r_k \cdot j^k$, where $\epsilon_j = \sum_{i=1}^{t+1} \epsilon_{ij}$, $r_k = \sum_{i=1}^{t+1} r_{ki}$ (notice that $r_0 = 0$). This can be written as $s'_j = \gamma_j + h(j)$, where $h(x) = f(x) + g(x) \in \mathbb{F}_{\leq t}[x]$, $g(x) = \sum_{k=0}^{t} r_k \cdot x^k \in \mathbb{F}_{\leq t}[x]$ and $\gamma_j = \epsilon_j + \delta_j$.

From the above it follows that the final sharings $s'_j = \gamma_j + h(j)$ output by the honest parties $P'_j \in C' \setminus A'$ are consistent: The adversary knows all γ_i, so it can re-define $s'_j \leftarrow s'_j - \gamma_j + q(j)$ for $P'_j \in \mathcal{A}'$, where $q(j) \in \mathbb{F}_{\leq t}[x]$ is such that $q(i) = \gamma_i$ for $P_i \in C' \setminus A'$, and in this way the sharings $(s'_1, \ldots, s'_j)$ are consistent with the polynomial $h(x) + q(x) \in \mathbb{F}_{\leq t}[x]$. Furthermore, if all parties behave honestly then $q(x) \equiv 0$, so the shares $(s'_1, \ldots, s'_n)$ are consistent with the polynomial $h(x)$ which satisfies $h(0) = f(0) + g(0) = s + 0 = s$.

Finally, we show that privacy holds. To see this, it suffices to show that $(s'_1 - s_1, \ldots, s'_n - s_n)$ are uniform shares of some value that the adversary knows. We first claim that, fromt the point of view of the adversary, $(g(1), \ldots, g(n))$ are uniformly random shares of 0. This holds because $s'_j - s_j = (s_j + q(j) + g(j)) - s_j = q(j) + g(j)$

Now we argue privacy. For this we assume that $q(x) \equiv 0$ (that is, the adversary did not cheat overall). This simplifies notation, but it is also without loss of generality because as we saw above the worst thing an adversary can do is shifting the secret by an amount the adversary itself knows. First, notice that the view of the adversary is

$$(\underbrace{\{s_{ij}\}_{P_i \in A, P_j \in U}, \{g_i(x)\}_{P_i \in U \cap A}}_{\text{Sampled locally}}, \underbrace{\{s_{ij}\}_{P_i \in C, P_j \in U \cap A}}_{\text{Received in step 1}}, \underbrace{\{a_{ij}\}_{P_i \in U, P'_j \in A'}}_{\text{Received in step 4}}),$$

where $g_i(x) = \sum_{k=0}^{t} r_{ki} \cdot x^t$ (notice that $g(x) = \sum_{i=1}^{t+1} g_i(x)$). We claim that this view is independent of the secret s. To see this, we define a simulator § that, on input $(\{s_{ij}\}_{P_i \in A, P_j \in U}, \{g_i(x)\}_{P_i \in U \cap A})$ and without knowledge of s, produces an indistinguishable view.

The simulator § is defined as follows:

- Sample $\mathsf{s}_{ij} \in_R \mathbb{F}$ for $P_i \in C \setminus A, P_j \in U \cap A$, and set $\mathsf{s}_{ij} := s_{ij}$ for $P_i \in A, P_j \in U \cap A$.
- Define $\mathsf{a}_{ij} := \mathsf{s}_{ji} + g_i(j)$ for $P_i \in U \cap A, P'_j \in A'$, and $\mathsf{a}_{ij} \in_R \mathbb{F}$ for $P_i \in U \setminus A, P'_j \in A'$
- Output

$$(\{\mathsf{s}_{ij}\}_{P_i \in A, P_j \in U}, \{\mathsf{g}_i(x)\}_{P_i \in A}, \{\mathsf{s}_{ij}\}_{P_i \in C, P_j \in U \cap A}, \{\mathsf{a}_{ij}\}_{P_i \in U, P'_j \in A'}).$$

The two views are perfectly indistinguishable: $\{\mathsf{s}_{ij}\}_{P_i \in C, P_j \in U \cap A} \equiv \{\mathsf{s}_{ij}\}_{P_i \in C, P_j \in U \cap A}$ because, given that $|U \cap A| \leq t < t+1$, in the real execution the honest parties $P_i \in C \setminus A$ sample $\{s_{ij}\}_{P_j \in U \cap A}$ independently and uniformly at random, like in the simulation. Also $\{a_{ij}\}_{P_i \in U, P'_j \in A'} \equiv \{\mathsf{a}_{ij}\}_{P_i \in U, P'_j \in A'}$ given the rest of the views because, in the real execution, $\{a_{ij}\}_{P_i \in U \setminus A, P'_j \in A'}$ are uniformly random since they are only conditioned on $a_j = \sum_{i=1}^{t+1} a_{ij} = s_j + g(j)$ for $P'_j \in A'$, but since $|A'| \leq t$ and $g(x) \in_R \mathbb{F}_{\leq t}[x]$ with $g(0) = 0$, $\{g(j)\}_{P'_j \in A'}$ are independent and uniform so $\{a_j\}_{P_j \in A'}$ look uniform and independent to the adversary. □ □

Proof of Theorem 3

Sketch. We only provide a sketch of the corresponding simulation-based proof. Let $s' = s + \delta$ and $\sigma'_2 = \sigma_1 + \gamma$, where $\delta \in \mathbb{F}$ and $\gamma \in \mathbb{G}_1$ are the errors introduced by the adversary in the $\Pi_{\mathsf{PartialPSS}}$ protocol. Our simulator simply emulates the role of the honest parties, with these virtual honest parties using random shares as inputs. The simulator also emulates all the necessary functionalities like $\mathcal{F}_{\mathsf{DotProduct}}$, $\mathcal{F}_{\mathsf{Coin}}$ and $\mathcal{F}_{\mathsf{Rand}}$. Using an argument along the lines of the proof of Theorem 2, the simulator is then able to learn the errors δ and γ. The simulator then makes the virtual parties abort if $\delta \neq 0$ or $\gamma \neq 0_{\mathbb{G}_1}$.

We show that the simulated execution is indistinguishable to the adversary from a real execution. To see this, first observe that in the real execution, the honest parties abort if the output of Verify^* is not 0. Furthermore, it is easy to see that the output of $\Pi_{\mathsf{Verify}}([\![s']\!]^{C'}, (\sigma_1, [\![\sigma_2']\!]^{C'}), \mathsf{pk}_C)$ is equal to 0 if and only if $\delta \cdot e(\sigma_1, Y) = e(\gamma, H)$. Given this, the only scenario in which the two executions (real and simulated) could differ is if $\delta \neq 0$ or $\gamma \neq 0_{\mathbb{G}_1}$, but $\delta \cdot e(\sigma_1, Y) = e(\gamma, H)$, since in this case the honest parties in the real execution do not abort, but the honest parties in the ideal execution do. However, we show this cannot happen: If $\delta \neq 0$ or $\gamma \neq 0_{\mathbb{G}_1}$, then $\delta \cdot e(\sigma_1, Y) \neq e(\gamma, H)$, with overwhelming probability.

To see why the claim above holds, we make a reduction to the co-CDH problem defined above: An adversary gets challenged with $(\alpha_1 H, \alpha_2 H')$, and its goal is to find $\alpha_1\alpha_2 H$. The adversary then plays the simulator above, but uses $\sigma_1 = \alpha_1 H$ and $Y = \alpha_2 H'$. Now suppose that in the simulation $\delta \neq 0$ and $\delta \cdot e(\sigma_1, Y) = e(\gamma, H')$. We can see then that this equation implies that $\delta\alpha_1\alpha_2 = \beta$, where $\beta \in \mathbb{F}$ is such that $\gamma = \beta H'$. In particular, it implies that $\alpha_1\alpha_2 H = \delta^{-1}\beta H = \delta^{-1}\gamma$, so the adversary, who knows δ and γ, can compute $\alpha_1\alpha_2 H$ as above, thus breaking co-CDH. Finally, it is easy to see that if $\gamma \neq 0$ and $\delta \cdot e(\sigma_1, Y) = e(\gamma, H)$, then $\delta \neq 0$ with high probability since otherwise $e(\gamma, H) = 0$, so the same argument as above works. This finishes the sketch of the simulation-based proof of the theorem. □ □

Shamir Secret-Sharing

Consider a setting with n parties, and let $0 < t < n$. In this scheme each party P_i gets $f(i)$ where $f(x) \in_R \mathbb{F}_{\leq t}[x]$ subject to $f(0) = s$, and $s \in \mathbb{F}$ is the secret.[8] We denote $[\![s]\!]_{\mathbb{F}}^{\mathsf{shm}} = (f(1), \ldots, f(n))$. More formally, this scheme $\mathcal{S}_{\mathsf{shm}}$ is defined as $(M_{\mathsf{shm}}, \mathsf{label}_{\mathsf{shm}})$, where $M_{\mathsf{shm}} \in \mathbb{F}^{n\times(t+1)}$ is given below, and $\mathsf{label}_{\mathsf{shm}}(i) = i$:

$$\begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_{n-1} \\ s_n \end{pmatrix} = \underbrace{\begin{pmatrix} 1^0 & 1^1 & 1^2 & \cdots & 1^t \\ 2^0 & 2^1 & 2^2 & \cdots & 2^t \\ & & \vdots & & \\ (n-1)^0 & (n-1)^1 & (n-1)^2 & \cdots & (n-1)^t \\ n^0 & n^1 & n^2 & \cdots & n^t \end{pmatrix}}_{M_{\mathsf{shm}} \in \mathbb{F}^{n\times(t+1)}} \cdot \begin{pmatrix} s \\ r_1 \\ r_2 \\ \vdots \\ r_t \end{pmatrix}$$

It is easy to see that this scheme is $(n-1, n)$-secure. Over a vector space V, sharing a point $\alpha \in V$ is done by sampling $r_1, \ldots, r_t \in_R V$, and setting the i-th share to be $\alpha_i = \alpha + \sum_{j=1}^t i^j \cdot r_j$. In this way, $\alpha_i = f(i)$, where $f(x) = \alpha + \sum_{j=1}^t x^j \cdot r_j \in_R V_{\leq t}[x]$. We denote this by $[\![S]\!]_V^{\mathsf{shm}}$.

[8] We assume that $|\mathbb{F}| > n + 1$

Reconstruction

Consider a shared value $[\![s]\!]^{\mathsf{shm}} = (f(1), \ldots, f(n))$. If $t \geq n/2$, then it can be shown that the adversary can succeed in opening an incorrect value by modifying the shares of the corrupt parties. However, if $t < n/2$, this cannot be done: The honest parties will be able to *detect* that the opened value is not correct. Furthermore, if $t < n/3$, the honest parties can do better: On top of detecting whether the open value is the right one, they can *correct* the errors and compute the right secret. We describe these below, and we also discuss extensions to elliptic curves.

Error detection ($t < n/2$). Assume $t < n/2$, and suppose that a most t shares among $(s_1, \ldots, s_n)$ are incorrect. If all shares $(s_1, \ldots, s_n)$ lie in a polynomial of degree at most t, then the reconstructed secret must be correct, given that a polynomial of degree at most t is determined by *any* $t+1$ points, in particular, it is determined by the $t+1 \leq n-t$ correct shares. In this way, by verifying if all the shares lie in a polynmial of the right degree, the parties can detect whether the reconstructed value is correct or not. This can be done by interpolating a polynomial of degree at most t using the first $t+1$ shares, and then checking whether the other shares are consistent with this polynomial.

Alternatively, the parties can use the *parity check matrix* $H \in \mathbb{F}^{(n-t-1)\times n}$, which satisfies that $A \cdot (s_1, \ldots, s_n)^T$ is the zero-vector if and only if the shares s_i are consistent with a polynomial of degree at most t. This check can be performed for the group sharings $[\![P]\!]_{\mathbb{G}}$ as well.

Error correction ($t < n/3$). If $t < n/2$ then the parties can detect whether a reconstructed value is correct or not, but they cannot "fix" the errors in case the value is not correct. Under the additional condition $t < n/3$, this can actually be done, that is, the parties can reconstruct the correct value, regardless of any changes the adversary does to the shares from corrupted parties. The algorithm to achieve this proceeds, at least conceptually, as follows: The parties find a subset of $2t+1$ shares among the announced shares that lies in a polynomial of degree at most t; this set exists because there are at least $n - t \geq 2t+1$ correct shares. Then, the secret given by this polynomial is taken as the right secret. This is correct because of the same reason as in the previous case: This polynomial is determined by any set of $t+1$ points among the $2t+1$ ones that are consistent, and in particular, it is determined by the $t+1 = 2t+1-t$ correct shares, since at most t of them can be incorrect.

The main bottleneck in the reconstruction algorithm sketched above is finding a consistent subset of $2t+1$ shares, since there are exponentially-many such sets. To this end, an error-correction algorithm like Berlekamp Welch is used [77], which has a running time that is polynomial in n.

Finally, it is important to remark that, unlike the error-detection mechanism above, this error-correction procedure *cannot* be performed over the group $\mathbb{G}$. This interesting result was shown in [125].

Dot Products of Shared Vectors

Let $2t+1=n$, and let U, V, W be $\mathbb{F}$-vector spaces of dimension d with bases $\{u_i\}_{i=1}^d$, $\{v_i\}_{i=1}^d$ and $\{w_i\}_{i=1}^d$, respectively.[9] Consider a bilinear map $\phi: U \times V \to W$. For the rest of this section we consider Shamir secret sharing, and we omit the superscript shm from the sharings, and consider explicitly the degree of the polynomial used for the sharing: $[\![\cdot]\!]^h$ denotes Shamir secret sharing using polynomials of degree at most h.

Consider shared values $[\![x_1]\!]_U^t, \dots, [\![x_L]\!]_U^t, [\![y_1]\!]_V^t, \dots, [\![y_L]\!]_V^t$. In this section we describe a protocol to compute $[\![z+\delta]\!]_W^t$, where $z = \sum_{\ell=1}^L \phi(x_\ell y_\ell)$ and $\delta \in W$ is some error known to the adversary. The main building blocks of the protocol are the following:

- The parties can locally obtain $[\![\phi(\alpha,\beta)]\!]_W^{2t}$ from $[\![\alpha]\!]_U^t$ and $[\![\beta]\!]_V^t$. To see this, write $[\![\alpha]\!]_U^t = (f(1),\dots,f(n))$ and $[\![\beta]\!]_U^t = (g(1),\dots,g(n))$, for some $f(x) \in U_{\leq t}[x]$ and $g(x) \in V_{\leq t}[x]$ such that $f(0)=\alpha$ and $g(0)=\beta$. Write $f(x) = \sum_{i=0}^t x^i \cdot r_i$ and $g(x) = \sum_{i=0}^t x^i \cdot s_i$, and let $h(x) = \sum_{i,j=1}^t x^{i+j} \cdot \phi(r_i, s_j) \in W_{\leq 2t}[x]$. It is easy to see that $h(0) = \phi(\alpha,\beta)$ and that $h(i) = \phi(f(i), g(i))$ for all $i = 1,\dots,n$, so $[\![\phi(\alpha,\beta)]\!]_W^{2t} = (h(1),\dots,h(n))$.
- There exists a protocol $\Pi_{\mathsf{DoubleShare}}$ that produces a pair $([\![w]\!]_W^t, [\![w]\!]_W^{2t})$, where $w \in_R W$. Such a pair can be produced from d pairs $([\![r_i]\!]_{\mathbb{F}}^t, [\![r_i]\!]_{\mathbb{F}}^{2t})$ by defining $[\![w]\!]_W^k = \sum_{i=1}^d [\![r_i]\!]^k \cdot w_i$ for $k = t, 2t$. These pairs over $\mathbb{F}$ can be produced using the protocol from [58].

With these tools at hand we are ready to describe our main protocol.

Protocol 12 Protocol $\Pi_{\mathsf{DotProduct}}^{\mathsf{shm}}$

Inputs: Shared values $[\![x_1]\!]_U, \dots, [\![x_L]\!]_U, [\![y_1]\!]_V, \dots, [\![y_L]\!]_V$.

Output: $[\![z+\delta]\!]_W$, where $z = \sum_{\ell=1}^L \phi(x_\ell, y_\ell)$ and $\delta \in W$ is some error known to the adversary.

1. Call $([\![w]\!]_W^t, [\![w]\!]_W^{2t}) \leftarrow \Pi_{\mathsf{DoubleShare}}$
2. Parties locally compute $[\![\phi(x_\ell, y_\ell)]\!]_W^{2t} \leftarrow \phi([\![x_\ell]\!]_U^t, [\![y_\ell]\!]_V^t)$, for $\ell = 1,\dots,L$;

[9] As in Section 5.2, the condition that all three spaces have the same dimension is not necessary.

3. Parties compute $[\![e]\!]_W = [\![w]\!]_W^{2t} + \sum_{\ell=1}^{L} [\![\phi(u_\ell, v_\ell)]\!]_W^{2t}$ and send the shares of e to P_1.

4. P_1 uses the $n = 2t + 1$ shares received to reconstruct $e + \delta$ (where δ is the error the adversary may introduce by lying about its shares), and broadcasts[10] $e + \delta$ to all parties.

5. All parties set $[\![z + \delta]\!]_W^t = (e + \delta) - [\![w]\!]_W^t$.

The protocol is private because the only value that is opened is e, which is a perfectly masked version of the sensitive value z, given that w is uniformly random and unknown to the adversary. The communication complexity of $\Pi^{\mathsf{shm}}_{\mathsf{DotProduct}}$ is $\mathcal{C}^{\mathsf{shm}}_{\mathsf{DotProd}} = d \cdot \log(|\mathbb{F}|) \cdot 5.5 \cdot n$, using the optimization from [88].

Optimizations

Optimizations for PSS

Optimizing the signatures. As we noted in Section 5.4, we can use the more efficient functionality $\mathcal{F}_{\mathsf{DotProduct}*}$ instead of $\mathcal{F}_{\mathsf{DotProduct}}$, at the expense of allowing the adversary to produce incorrect signatures by adding any error to the second component of the signature. However, this is completely acceptable in our setting. In fact, the adversary can already add an error to the second component of the signature when using the $\Pi_{\mathsf{PartialPSS}}$ protocol. Hence, in our protocol Π_{PSS} we use the modified version of Π_{Sign} that uses $\mathcal{F}_{\mathsf{DotProduct}*}$ instead of $\mathcal{F}_{\mathsf{DotProduct}}$.

Using AMD codes. The fact that the worst that can happen in the $\Pi_{\mathsf{PartialPSS}}$ protocol is that the transmitted message is wrong by an additive amount known by the adversary implies that other methods to ensure correctness of the transmitted value can be devised, like the MACs described in Section 5.3 for additive secret-sharing. Although the overall computation is much more efficient since it does not involve any public-key operations, the communication of the method we present here is worse by a factor of 2.

Optimizations for Input Certification

Optimization if multiple parties provide input. If all parties $P_1, \ldots, P_n$ use $\Pi_{\mathsf{CertInput}}$ to certify their input, each party can call $\Pi_{\mathsf{CertInput}}$, which, in the case that a protocol with guaranteed output delivery is used to compute Π_{Verify}, allows parties to identify exactly which party provided a faulty input. However, one can improve the communication complexity if a "global" abort is

accepted, that is, if the parties do not abort then *all* the inputs are correctly certified, but if they do abort, then it is not possible to identify which party provided an incorrect input (however, for protocols without guaranteed output delivery, this is acceptable since the abort can already happen due to malicious behavior in other parts of the protocol).

The optimization works as follows. Consider the n $\Pi_{\mathsf{CertInput}}$ executions, corresponding to all parties. At the end of step 2, n shares $[\![r_1]\!]_{\mathbb{G}_T}, \ldots, [\![r_n]\!]_{\mathbb{G}_T}$ have been produced. The parties then locally compute $[\![r]\!]_{\mathbb{G}_T} = \prod_{i=1}^{n} [\![r_i]\!]_{\mathbb{G}_T}$ (recall that $\mathbb{G}_T$ is a multiplicative group), open r, and accept the secret-shared inputs if and only if this opened value equals $1_{\mathbb{G}_T}$. Notice that, if at least one signature is incorrect, then at least one r_i is uniformly random, so r will be uniformly random too and therefore the probability that it equals $1_{\mathbb{G}_T}$ in this case is at most $1/|\mathbb{G}_T|$.

Communication Complexity of CHURP

CHURP, a dynamic PSS protocol proposed in [113], is the state of the art in terms of communication complexity. At a high level, CHURP is made of two main protocols, Opt-CHURP, which is able to detect malicious behavior during the proactivization but is not able to point out which party or parties cheated, and Exp-CHURP, which performs proactivization while enabling cheater detection at the expense of being heavier in terms of communication. Since in this work we have described a PSS protocol *with abort*, we compare our protocol against Opt-CHURP.

The protocol Opt-CHURP is comprised of three main subprotocols: Opt-ShareReduce, Opt-Proactivize and Opt-ShareDist. In the first sub-protocol, Opt-ShareReduce, the parties in C distribute shares of their shares towards the parties in C'. A threshold of $2t$ is used for these "two-level" shares to account for the fact that the adversary may control t parties in each committee C and C'. We could avoid such high degree sharing in our $\Pi_{\mathsf{PartialPSS}}$ protocol since there the parties do not share their shares directly. In Opt-ShareReduce, to ensure that a party sends the right share, the parties must also communicate commitments and witnesses for certain polynomial commitment scheme (see [113] for details). The concrete communication complexity of this step is $2Ln^2$ elements, where L is the amount of shared field elements being proactivized.

In the second stage, Opt-Proactivize the parties in C' produce reduced-shares (that is, "shares of shares") of 0 that are added to the reduce-shares of the secret. We will not discuss the details fo this procedure here, beyond mentioning that this requires the parties to exchange shares and proofs in order to ensure the correctness of this method. This incurs a communication complexity of $5Ln^2$ field elements, on top of requiring publishing n hashes on a blockchain, say $256n$ bits using SHA256, which is a requirement that our protocol Π_{PSS} does not have.

In the final stage, Opt-ShareDist, each party in C' sends the reduce-shares of the i-th share to party P'_i, who reconstructs the refreshed share. Again, opening information for certain commitments must be transmitted. This leads to a communication complexity of $2Ln^2$.

We see then that the total (off-chain) communication complexity in Opt-CHURP is $9Ln^2 \log(|\mathbb{F}|)$ bits.

Secure Computation over Elliptic Curves

So far we have presented a fairly comprehensive "toolbox" for performing secure computation over elliptic curves. We may view the LSS homomorphism $\phi : \mathbb{F}_p \to \mathbb{G}$ defined by $\phi(x) = x \cdot G$ as a function that encodes x into the exponent of G. While this enables the applications we presented in Section 5.4, Section 5.5 and Section 5.6, it does not an efficient way of *decoding*.

The following example illustrates why this might lead to issues in some applications: Parties hold $[\![m]\!]_{\mathbb{F}}$ and wish to encrypt it using El-Gamal. Using an LSS homomorphism on $[\![m]\!]$ would effectively encode m in the exponent, and then we could use secure computation over elliptic curves to compute the encryption of m.

The above works for encryption. But what if the parties wish to recover $[\![m]\!]$ from the encryption? Clearly, a party cannot recover m_i from $m_i \cdot G$ since m_i (the share) is a random field element. On the other hand, we cannot reconstruct $m \cdot G$ towards a party as that would reveal the message.

The issue above arises from the fact that the encoding of $[\![m]\!]$ was done using the LSS homomorphism $x \mapsto x \cdot G$, which is highly efficient due to its linearity, but has a "one-wayness" to it, making it very hard to decode. In the following, we show a different way of encoding a shared field element $[\![m]\!]$ in such a way that, although the encoding itself is interactive (and therefore less efficient than the LSS homomorphism encoding described above), the decoding process is practically efficient. This enables a seamless interplay between traditional secure computation over $\mathbb{F}$, and secure computation over an elliptic curve group as defined here.

Secure Encoding and Decoding

We now show how to map secret-shared messages into curve points, and back, in the presence of an active adversary and an honest majority. Consider the following commonly used injective encoding for encoding bit-strings into points on the curve $\mathbb{G}$ over $\mathbb{F}$ (see [74]): To encode a message $m \in \{0,1\}^\ell$, with $\ell \leq (1/2 - \epsilon)\log_2 p$ for a fixed $\epsilon \in (0, 1/2)$, pick a random integer $x \in [0, p-1]$ such that $m = x \mod 2^\ell$. If x is a valid curve-point for $\mathbb{G}$, then output (x, y), and otherwise pick a new random x and start over. We denote this encoding by En and its inverse as De (notice that De simple discards y and returns $x \mod 2^\ell$).

Our aim now is to implement $(\mathsf{En}, \mathsf{De})$ securely; that is, we wish to compute $[\![\mathsf{En}(x)]\!]$ given $[\![x]\!]$ with $x \in \{0,1\}^\ell$, and $[\![\mathsf{De}(X)]\!]$ given $[\![X]\!]_\mathbb{G}$ with $\mathsf{En}(m) = X \in \mathbb{G}$ for some m. For this we will use two functionalities: The first protocol is $\mathcal{F}_{\mathsf{IsSqr}}$, which takes as input a secret-shared value $[\![x]\!]$ and outputs 1 if x is a square, and 0 otherwise. That is, if $\mathcal{F}_{\mathsf{IsSqr}}$ outputs 1, then there exists a value y such that $x^2 = y \mod p$. The other protocol is $\mathcal{F}_{\mathsf{Sqrt}}$ which, on input a square $[\![x]\!]$, outputs $[\![y]\!]$ satisfying $y = x^2 \bmod p$.[11]

In the following, we assume that the curve is given as $y^2 = x^3 + ax + b$ where a and b are constants.

Decoding. We begin with decoding. Given a secret-sharing $[\![\mathsf{En}(m)]\!]_\mathbb{G}$ where $\mathsf{En}(m) = (x, y)$ and $m \equiv x \mod 2^\ell$, the goal is to obtain $[\![m]\!]$. Besides $[\![\mathsf{En}(m)]\!]_\mathbb{G}$, we assume that we also have access to a secret-sharing of the upper $\ell - \log_2 p$ bits of x and we denote this value as $[\![r]\!]$. Write $[\![z]\!]_\mathbb{G} = [\![\mathsf{En}(m)]\!]_\mathbb{G}$ and let x_i, resp. y_i be the values that comprise the i'th party's share of z. To decode z, each party first re-shares the x_i and y_i they hold, after which everyone computes the point addition formula over all the coordinates. In a nutshell, this is the same idea used when decomposing a number into bits. In this scenario, parties mask the value they want to bit-decompose and then compute a binary adder to unmask each bit.

Protocol 13 Protocol Π_{Decode}

Inputs: $[\![X]\!]_\mathbb{G}$, $[\![r]\!]$ where r was the randomness added during encoding.

Outputs: $[\![m]\!]$ the encoded message, secret-shared over the basefield.

1. Each party P_i parses their share of $[\![X]\!]_\mathbb{G}$ as the pair (x_i, y_i) and secret-shares $[\![x_i]\!]$, $[\![y_i]\!]$ towards the other parties.
2. Parties apply a parity check matrix to check that the reshared values are consistent (see Appendix 5.7 for details).
3. For $j = 2, \ldots, t+1$ where t is the number of corrupt parties, compute the curve addition of the shares over the secret-shared coordinates:
   a) Invoke $[\![a]\!] = \mathcal{F}_{\mathsf{Rand}}(\mathbb{F})$.
   b) $[\![z]\!] \leftarrow \mathcal{F}_{\mathsf{Mult}}([\![x_j - x_{j-1}]\!], [\![a]\!])$ and open z.
   c) Compute $[\![d]\!] = [\![(x_j - x_{j-1})^{-1}]\!] = z^{-1}[\![a]\!]$, $[\![\lambda]\!] = \mathcal{F}_{\mathsf{Mult}}([\![y_j - y_{j-1}]\!], [\![d]\!])$ and finally $[\![\lambda^2]\!] = \mathcal{F}_{\mathsf{Mult}}([\![\lambda]\!], [\![\lambda]\!])$.
   d) Compute $[\![x']\!] = [\![\lambda^2]\!] - [\![x_j]\!] - [\![x_{j-1}]\!]$.

---

[11] We show how to instantiate these in the full version of this paper. Put briefly, $\mathcal{F}_{\mathsf{IsSqr}}$ is straightforward when $p \equiv 3 \bmod 4$, and $\mathcal{F}_{\mathsf{Sqrt}}$ can be implemented with a trick from [15] that enables computing $[\![x^{-1}]\!]$ given $[\![x]\!]$ and a multiplication triple.

e) Compute $[\![y'']\!] = \mathcal{F}_{\mathsf{Mult}}([\![\lambda]\!], [\![x_j - x']\!])$ and $[\![y']\!] = [\![y'']\!] - [\![y_j]\!]$.
f) Set $[\![x_j]\!] = [\![x']\!]$ and $[\![y_j]\!] = [\![y']\!]$.

4. Output $[\![x_{t+1}]\!] - [\![r]\!]$.

Protocol Π_{Decode} computes the injective encoding with complexity $\mathcal{C}_{\mathsf{Share}}(n) + \mathcal{C}_{\mathsf{Check}}(n) + (t+1)(\mathcal{C}_{\mathsf{Rand}}(1) + \mathcal{C}_{\mathsf{Mult}}(4) + \mathcal{C}_{\mathsf{Open}}(1))$.

Lemma 9. *Protocol Π_{Decode} securely outputs the lower ℓ bits of $[\![X]\!]_{\mathbb{G}}$.*

Proof. Let $X_i = (x_i, y_i)$ be the i'th party's share of $X = (x, y)$. Notice that X can be reconstructed as a linear combination of the X_i's; in particular, $X = \sum_{i=1}^{t+1} X_i$ (we omit constants in this linear combination for the sake of simplicity). This linear combination is computed in step 3 in the protocol, so, at step 3.f, parties hold shares of the coordinates of X, secret-shared over the base field. Finally, $[\![x]\!] - [\![r]\!]$ removes the randomness located in the upper $\log_2 p - \ell$ bits of x. Step 1 potentially poses a problem, as a corrupt party may secret-share an incorrect value. However, the parity check applied in step 2 ensures this cannot happen, as the adversary can only modify at most t shares. □ □

Encoding. To encode a value $x \in \mathbb{F}$, recall that we first need to add a bit of randomness to it, in order to have a chance at hitting a valid x-coordinate for our curve. Let ℓ be an upper bound on the size of x, i.e., $x \leq 2^\ell$. We first consider a straightforward, but ultimately insecure, approach utilizing $\mathcal{F}_{\mathsf{Coin}}$: Parties use $\mathcal{F}_{\mathsf{Coin}}$ to sample a random value $r < p$ such that its lower ℓ bits are 0. Parties then call $\mathcal{F}_{\mathsf{IsSqr}}([\![x]\!] + r)$, and restart the process (i.e., go back and pick another r) if this protocol outputs 0. However this fails to be secure. Indeed, if x is of low entropy, then revealing whether or not $[\![x]\!] + r$ is a square, reveals information about x itself (in particular, the adversary can rule out values x' for which $x' + r$ *is* a square).

We must thus resort to fancier machinations that allows us to sample an appropriate r without revealing it. Luckily, sampling a random value where its lower bits are zero has been used before—in particular in connection with secure truncation protocols (see e.g., [54]). We thus assume a functionality $\mathcal{F}_{\mathsf{sRand}}$ which outputs a secret-shared r suitable for our purposes. The final thing we require is a tuple $([\![R]\!]_{\mathbb{G}}, [\![r_x]\!], [\![r_y]\!])$ where $R = (r_x, r_y)$. Such a tuple can be generated by sampling a random $[\![R]\!]_{\mathbb{G}}$ and then using step 2 in Π_{Decode} to obtain $[\![r_x]\!]$ and $[\![r_y]\!]$.

Protocol 14 Protocol Π_{Encode}

Inputs: $[\![m]\!]$ the message to be encoded.

Outputs: $[\![\mathsf{En}(m)]\!]_{\mathbb{G}}$, $[\![r]\!]$.

1. Sample $[\![r]\!] = \mathcal{F}_{\mathsf{sRand}}$ and compute $[\![x]\!] = [\![m]\!] + [\![r]\!]$.
2. Call $\mathcal{F}_{\mathsf{IsSqr}}([\![x]\!])$. If the return value is 0, go back to the previous step.
3. Call $[\![y]\!] = \mathcal{F}_{\mathsf{Sqrt}}([\![x^3]\!] + [\![x]\!]a + b)$. Note that parties now have $[\![x]\!]$, $[\![y]\!]$ which are secret-sharings of $\mathsf{En}(m)$ in the field.
4. Parties then compute the curve addition formula between the points $([\![x]\!], [\![y]\!])$ and $([\![r_x]\!], [\![r_y]\!])$. Let $([\![z_x]\!], [\![z_y]\!])$ be the result.
5. $[\![z_x]\!]$ and $[\![z_y]\!]$ is opened. Write $Z = (z_x, z_y)$.
6. Output $[\![\mathsf{En}(m)]\!]_{\mathbb{G}} = [\![X]\!]_{\mathbb{G}} = Z - [\![R]\!]_{\mathbb{G}}$ and $[\![r]\!]$.

Protocol Π_{Encode} computes the injective encoding of m with complexity

$$\mathcal{C}_{\mathsf{Encode}} = \mathcal{C}_{\mathsf{sRand}}(k) + \mathcal{C}_{\mathsf{IsSqr}}(k) + \mathcal{C}_{\mathsf{Sqrt}}(1) + 2\mathcal{C}_{\mathsf{Open}}(1) + \mathcal{C}_{\mathsf{Rand}}(1) + \mathcal{C}_{\mathsf{Mult}}(4).$$

Security comes from the fact that, at the end of step 5, parties hold $Z = X + R$, and since R is random, nothing is revealed about X. In the cost formula, k denotes the number of repetitions of the first two steps. [74] proves that a suitable r is found in expected 3 iterations (i.e., k has expected value 3).

Cats or Croissants?

Figure 1: Cats or Croissants?

Bibliography

[1] Partisia. URL https://partisia.com/. 5

[2] Sepior. URL https://sepior.com/.

[3] Unbound tech. URL https://www.unboundtech.com/. 5

[4] Fast large-scale honest-majority mpc for malicious adversaries, 2017. https://github.com/cryptobiu/MPCHonestMajorityNoTriples. 36, 37

[5] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. Efficient information-theoretic secure multiparty computation over $\mathbb{Z}/p^k\mathbb{Z}$ via galois rings. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 471–501. Springer, Heidelberg, December 2019. doi: 10.1007/978-3-030-36030-6_19. 12, 16, 20, 21, 24, 30, 31, 32, 34

[6] Mark Abspoel, Anders Dalskov, Daniel Escudero, and Ariel Nof. An efficient passive-to-active compiler for honest-majority mpc over rings. Cryptology ePrint Archive, Report 2019/1298, 2019. https://eprint.iacr.org/2019/1298. 11

[7] Abdelrahaman Aly, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Nigel P Smart, and Tim Wood. Scale–mamba v1. 3: Documentation. Technical report, Technical Report, 2019. 44

[8] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, October 2016. doi: 10.1145/2976749.2978331. 8, 12, 75, 96, 105

[9] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security*

and Privacy, pages 843–862. IEEE Computer Society Press, May 2017. doi: 10.1109/SP.2017.15. 5, 22

[10] D. F. Aranha, C. P. L. GouvÃªa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. `https://github.com/relic-toolkit/relic`. 87

[11] Diego Aranha, Anders Dalskov, Daniel Escudero, and Claudio Orlandi. Lss homomorphisms and applications to secure signatures, proactive secret sharing and input certification. Cryptology ePrint Archive, Report 2020/691, 2020. `https://eprint.iacr.org/2020/691`. 11, 15

[12] Marshall Ball, Tal Malkin, and Mike Rosulek. Garbling gadgets for Boolean and arithmetic circuits. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 565–577. ACM Press, October 2016. doi: 10.1145/2976749.2978410. 104

[13] Marshall Ball, Brent Carmer, Tal Malkin, Mike Rosulek, and Nichole Schimanski. Garbled neural networks are practical. Cryptology ePrint Archive, Report 2019/338, 2019. `https://eprint.iacr.org/2019/338`. 104

[14] Wolfgang Balzer, Masanobu Takahashi, Jun Ohta, and Kazuo Kyuma. Weight quantization in boltzmann machines. *Neural Networks*, 4(3): 405–409, 1991. 48

[15] Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In Piotr Rudnicki, editor, *8th ACM PODC*, pages 201–209. ACM, August 1989. doi: 10.1145/72981.72995. 118

[16] Mauro Barni, Claudio Orlandi, and Alessandro Piva. A privacy-preserving protocol for neural-network-based computation. In *Proceedings of the 8th workshop on Multimedia and security*, pages 146–151. ACM, 2006. 103

[17] Solon Barocas, Moritz Hardt, and Arvind Narayanan. *Fairness and Machine Learning.* fairmlbook.org, 2019. `http://www.fairmlbook.org`. 14

[18] Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. How to withstand mobile virus attacks, revisited. In Magnús M. Halldórsson and Shlomi Dolev, editors, *33rd ACM PODC*, pages 293–302. ACM, July 2014. doi: 10.1145/2611462.2611474. 72

[19] Joshua Baron, Karim El Defrawy, Joshua Lampkins, and Rafail Ostrovsky. Communication-optimal proactive secret sharing for dynamic groups. In

Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 15*, volume 9092 of *LNCS*, pages 23–41. Springer, Heidelberg, June 2015. doi: 10.1007/978-3-319-28166-7_2. 72, 83

[20] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, Heidelberg, August 1992. doi: 10.1007/3-540-46766-1_34. 7

[21] Zuzana Beerliová-Trubíniová and Martin Hirt. Efficient multi-party computation with dispute control. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 305–328. Springer, Heidelberg, March 2006. doi: 10.1007/11681878_16. 20

[22] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *20th ACM STOC*, pages 1–10. ACM Press, May 1988. doi: 10.1145/62212.62213. 4

[23] Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 663–680. Springer, Heidelberg, August 2012. doi: 10.1007/978-3-642-32009-5_39. 20

[24] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society Press, May 2014. doi: 10.1109/SP.2014.36. 87

[25] Marina Blanton and Fattaneh Bayatbabolghani. Efficient server-aided secure two-party function evaluation with applications to genomic computation. *PoPETs*, 2016(4):144–164, October 2016. doi: 10.1515/popets-2016-0033. 72

[26] Marina Blanton and Myoungin Jeong. Improved signature schemes for secure multi-party computation with certified inputs. In Javier López, Jianying Zhou, and Miguel Soriano, editors, *ESORICS 2018, Part II*, volume 11099 of *LNCS*, pages 438–460. Springer, Heidelberg, September 2018. doi: 10.1007/978-3-319-98989-1_22. 14, 72, 88, 89

[27] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206.

Springer, Heidelberg, October 2008. doi: 10.1007/978-3-540-88313-5_13. 22

[28] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *FC 2009*, volume 5628 of *LNCS*, pages 325–343. Springer, Heidelberg, February 2009. 4, 8

[29] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *CoRR*, abs/1604.07316, 2016. URL `http://arxiv.org/abs/1604.07316`. 42

[30] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 67–97. Springer, Heidelberg, August 2019. doi: 10.1007/978-3-030-26954-8_3. 37, 100

[31] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 483–512. Springer, Heidelberg, August 2018. doi: 10.1007/978-3-319-96878-0_17. 47, 103

[32] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 869–886. ACM Press, November 2019. doi: 10.1145/3319535.3363227. 6, 22, 37

[33] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. 3

[34] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. Flash: Fast and robust framework for privacy-preserving machine learning. Cryptology ePrint Archive, Report 2019/1365, 2019. `https://eprint.iacr.org/2019/1365`. 9

[35] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000. doi: 10.1007/s001459910006. 22

[36] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001. doi: 10.1109/SFCS.2001.959888. 6

[37] Ignacio Cascudo, Ronald Cramer, Chaoping Xing, and Chen Yuan. Amortized complexity of information-theoretically secure MPC revisited. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 395–426. Springer, Heidelberg, August 2018. doi: 10.1007/978-3-319-96878-0_14. 32

[38] Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 182–199. Springer, Heidelberg, September 2010. doi: 10.1007/978-3-642-15317-4_13. 55, 60

[39] Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 35–50. Springer, Heidelberg, January 2010. 60

[40] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. Cryptology ePrint Archive, Report 2017/035, 2017. `http://eprint.iacr.org/2017/035`. 103

[41] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. ASTRA: high throughput 3pc over rings with application to secure prediction. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop, CCSW@CCS 2019, London, UK, November 11, 2019*, pages 81–92, 2019. doi: 10.1145/3338466.3358922. URL `https://doi.org/10.1145/3338466.3358922`. 21, 22, 37

[42] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. In *NDSS 2020*. The Internet Society, February 2020. 4

[43] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *20th ACM STOC*, pages 11–19. ACM Press, May 1988. doi: 10.1145/62212.62214. 4

[44] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. URL `http://arxiv.org/abs/1512.01274`. 43

[45] 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 Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 34–64. Springer, Heidelberg, August 2018. doi: 10.1007/978-3-319-96878-0_2. 12, 20, 21, 24, 26, 27, 29, 36, 37, 38, 39, 97

[46] Boston Women's Workforce Council. Ensuring secure and private data analyses. 8

[47] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016. URL `http://arxiv.org/abs/1602.02830`. 47, 48

[48] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015. 48

[49] Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 316–334. Springer, Heidelberg, May 2000. doi: 10.1007/3-540-45539-6_22. 70

[50] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure multiparty computation*. Cambridge University Press, 2015. 73

[51] 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 Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Heidelberg, August 2018. doi: 10.1007/978-3-319-96881-0_26. 20, 22, 26, 46, 104, 105

[52] Morten Dahl, Jason Mancuso, Yann Dupis, Ben Decoste, Morgan Giraud, Ian Livingstone, Justin Patriquin, and Gavin Uhma. Private machine learning in tensorflow using secure computation. *CoRR*, abs/1810.08130, 2018. URL `http://arxiv.org/abs/1810.08130`. 4, 42

[53] Anders Dalskov, Daniele Lain, Enis Ulqinaku, Kari Kostiainen, and Srdjan Capcun. 2fe: Two-factor encryption for cloud storage. Currently in submission. The work presents a method for secure cloud storage that simultaniously provides a notion of two-factor security, but also allows the user to recover access to their files should one of their devices get lost. 11, 16

[54] Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *Proceedings on Privacy Enhancing Technologies*, 2020(4):355 – 375, 01 Oct. 2020. doi: https://doi.org/10.2478/popets-2020-0077. URL `https://content.sciendo.com/view/journals/popets/2020/4/article-p355.xml`. 8, 9, 11, 14, 15, 20, 119

[55] Anders Dalskov, Eysa Lee, and Eduardo Soria-Vazquez. Circuit amortization friendly encodings and their application to statistically secure multiparty computation. Cryptology ePrint Archive, Report 2020/1053, 2020. `https://eprint.iacr.org/2020/1053`. Note: published in ASIACRYPT 2020. 11, 15

[56] Anders Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. Securing dnssec keys via threshold ecdsa from generic mpc. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve Schneider, editors, *Computer Security – ESORICS 2020*, pages 654–673, Cham, 2020. Springer International Publishing. ISBN 978-3-030-59013-0. 11, 15, 70, 71, 72

[57] Anders P. K. Dalskov and Claudio Orlandi. Can you trust your encrypted cloud?: An assessment of SpiderOakONE's security. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *ASIACCS 18*, pages 343–355. ACM Press, April 2018. 11

[58] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Heidelberg, August 2007. doi: 10.1007/978-3-540-74143-5_32. 7, 32, 34, 72, 75, 85, 114

[59] 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 Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013. doi: 10.1007/978-3-642-40203-6_1. 20, 72, 75

[60] Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 799–829. Springer, Heidelberg, August 2018. doi: 10.1007/978-3-319-96881-0_27. 21

[61] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. Cryptology

ePrint Archive, Report 2019/599, 2019. https://eprint.iacr.org/2019/599. 53

[62] 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*, pages 1102–1120. IEEE Computer Society Press, May 2019. doi: 10.1109/SP.2019.00078. 20, 22, 46, 55, 56, 104

[63] Data61. MP-SPDZ. https://github.com/data61/MP-SPDZ, 2020. 64

[64] Alexandre de Brébisson and Pascal Vincent. An exploration of softmax alternatives belonging to the spherical loss family. *arXiv preprint arXiv:1511.05042*, 2015. 60

[65] 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. 103

[66] Jared A Dunnmon, Darvin Yi, Curtis P Langlotz, Christopher Ré, Daniel L Rubin, and Matthew P Lungren. Assessment of convolutional neural networks for automated classification of chest radiographs. *Radiology*, 290(2):537–544, 2018. 42, 43

[67] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! Arithmetic 3PC for any modulus with active security. In Yael Tauman Kalai, Adam D. Smith, and Daniel Wichs, editors, *ITC 2020*, pages 5:1–5:24. Schloss Dagstuhl, June 2020. doi: 10.4230/LIPIcs.ITC.2020.5. 12, 20, 21, 22, 37, 39, 40, 105

[68] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115, 2017. 3, 4, 42, 43

[69] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org. 43

[70] Facebook. CrypTen: a framework for privacy preserving machine learning. https://github.com/facebookresearch/CrypTen. 4, 42

[71] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th FOCS*, pages 427–437. IEEE Computer Society Press, October 1987. doi: 10.1109/SFCS.1987.4. 70

[72] Emile Fiesler, Amar Choudry, and H John Caulfield. Weight discretization paradigm for optical neural networks. In *Optical interconnections and networks*, volume 1281, pages 164–174. International Society for Optics and Photonics, 1990. 48

[73] Dario Fiore and Rosario Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 501–512. ACM Press, October 2012. doi: 10.1145/2382196.2382250. 85

[74] Pierre-Alain Fouque, Antoine Joux, and Mehdi Tibouchi. Injective encodings to elliptic curves. In Colin Boyd and Leonie Simpson, editors, *ACISP 13*, volume 7959 of *LNCS*, pages 203–218. Springer, Heidelberg, July 2013. doi: 10.1007/978-3-642-39059-3_14. 117, 120

[75] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255. Springer, Heidelberg, April / May 2017. doi: 10.1007/978-3-319-56614-6_8. 20, 21, 22

[76] Juan A Garay, Rosario Gennaro, Charanjit Jutla, and Tal Rabin. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1-2): 363–389, 2000. 70

[77] Peter Gemmell and Madhu Sudan. Highly resilient correctors for polynomials. *Information processing letters*, 43(4):169–174, 1992. 113

[78] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th ACM STOC*, pages 495–504. ACM Press, May / June 2014. doi: 10.1145/2591796.2591861. 20, 24

[79] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: From passive to active security via secure SIMD circuits. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 721–741. Springer, Heidelberg, August 2015. doi: 10.1007/978-3-662-48000-7_35. 24

[80] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In Maria-Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop*

and Conference Proceedings, pages 201–210. JMLR.org, 2016. URL http://proceedings.mlr.press/v48/gilad-bachrach16.html. 42, 43, 103

[81] Oded Goldreich. *Foundations of Cryptography: Basic Applications*, volume 2. Cambridge University Press, Cambridge, UK, 2004. ISBN ISBN 0-521-83084-2 (hardback). 22

[82] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987. doi: 10.1145/28395.28420. 4

[83] Robert E. Goldschmidt. Applications of division by convergence. Master's thesis, MIT, 1964. 60

[84] Shafi Goldwasser and Yehuda Lindell. Secure multi-party computation without agreement. *Journal of Cryptology*, 18(3):247–287, July 2005. doi: 10.1007/s00145-005-0319-z. 23

[85] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014. 48

[86] Google. Tensorflow lite. https://www.tensorflow.org/lite/, 2019. 49

[87] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional MPC with guaranteed output delivery. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part II*, volume 11693 of *LNCS*, pages 85–114. Springer, Heidelberg, August 2019. doi: 10.1007/978-3-030-26951-7_4. 35

[88] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority MPC. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 618–646. Springer, Heidelberg, August 2020. doi: 10.1007/978-3-030-56880-1_22. 22, 115

[89] Yunhui Guo. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752*, 2018. 50

[90] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. 48

[91] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure multi-party computation. In

2019 IEEE Symposium on Security and Privacy, pages 1220–1237. IEEE Computer Society Press, May 2019. doi: 10.1109/SP.2019.00028. 44

[92] Will Douglas Heaven. Openai's new language generator gpt-3 is shockingly good-and completely mindless, 07 2020. `https://www.technologyreview.com/2020/07/20/1005454/openai-machine-learning-language-generator-gpt-3-nlp/`. 3

[93] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In Don Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 339–352. Springer, Heidelberg, August 1995. doi: 10.1007/3-540-44750-4_27. 72

[94] Amir Herzberg, Markus Jakobsson, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive public key and signature systems. In Richard Graveman, Philippe A. Janson, Clifford Neuman, and Li Gong, editors, *ACM CCS 97*, pages 100–110. ACM Press, April 1997. doi: 10.1145/266420.266442. 72

[95] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017. URL `http://arxiv.org/abs/1704.04861`. 60

[96] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016. 47, 104

[97] ImageNet. Citations and publications. `http://image-net.org/about-publication`. 60

[98] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, pages 448–456, 2015. URL `http://jmlr.org/proceedings/papers/v37/ioffe15.html`. 49

[99] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877, 2017. URL `http://arxiv.org/abs/1712.05877`. 45, 49, 50, 51, 52, 54

[100] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 1651–1669. USENIX Association, August 2018. 42, 103

[101] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010. doi: 10.1007/978-3-642-17373-8_11. 70

[102] Jonathan Katz, Alex J. Malozemoff, and Xiao Wang. Efficiently enforcing input validity in secure two-party computation. Cryptology ePrint Archive, Report 2016/184, 2016. `http://eprint.iacr.org/2016/184`. 72

[103] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. Cryptology ePrint Archive, Report 2020/521, 2020. `https://eprint.iacr.org/2020/521`. 44

[104] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016. doi: 10.1145/2976749.2978357. 20, 104

[105] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, April / May 2018. doi: 10.1007/978-3-319-78372-7_6. 105

[106] Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 543–571. Springer, Heidelberg, August 2016. doi: 10.1007/978-3-662-53018-4_20. 87

[107] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. Swift: Super-fast and robust privacy-preserving machine learning. Cryptology ePrint Archive, Report 2020/592, 2020. `https://eprint.iacr.org/2020/592`. 9

[108] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018. 49

[109] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow: Secure TensorFlow inference. In *2020 IEEE Symposium on Security and Privacy*, pages 336–353. IEEE Computer Society Press, May 2020. doi: 10.1109/SP40000.2020.00092. 4, 8, 14, 42, 43, 47, 67, 104

[110] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew Back. Face recognition: A convolutional neural network approach. *Neural Networks, IEEE Transactions on*, 8:98 – 113, 02 1997. doi: 10.1109/72.554195. 42

[111] Yehuda Lindell and Ariel Nof. A framework for constructing fast MPC over arithmetic circuits with malicious adversaries and an honest-majority. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 259–276. ACM Press, October / November 2017. doi: 10.1145/3133956.3133999. 20, 96, 97, 105

[112] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. Oblivious neural network predictions via MiniONN transformations. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 619–631. ACM Press, October / November 2017. doi: 10.1145/3133956.3134056. 42, 47, 103

[113] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. CHURP: Dynamic-committee proactive secret sharing. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2369–2386. ACM Press, November 2019. doi: 10.1145/3319535.3363203. 14, 72, 82, 85, 116

[114] Michele Marchesi, Gianni Orlandi, Francesco Piazza, and Aurelio Uncini. Fast neural networks without multipliers. *IEEE transactions on Neural Networks*, 4(1):53–62, 1993. 48

[115] Alfred Menezes, Palash Sarkar, and Shashank Singh. Challenges with assessing the impact of NFS advances on the security of pairing-based cryptography. In *Mycrypt*, volume 10311 of *Lecture Notes in Computer Science*, pages 83–108. Springer, 2016. 87

[116] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020*, pages 2505–2522. USENIX Association, August 2020. 4, 9

[117] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018. doi: 10.1145/3243734.3243760. 9, 20, 37, 55, 56, 66, 67, 96, 103

[118] Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society Press, May 2017. doi: 10.1109/SP.2017.12. 4, 9, 20, 42, 60, 104

[119] Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority MPC by batchwise multiplication verification. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 321–339. Springer, Heidelberg, July 2018. doi: 10.1007/978-3-319-93387-0_17. 100

[120] Claudio Orlandi, Alessandro Piva, and Mauro Barni. Oblivious neural network computing via homomorphic encryption. *EURASIP Journal on Information Security*, 2007(1):037343, 2007. 103

[121] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In Luigi Logrippo, editor, *10th ACM PODC*, pages 51–59. ACM, August 1991. doi: 10.1145/112600.112605. 72

[122] Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. Weighted-entropy-based quantization for deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7197–7205. IEEE, 2017. 48

[123] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. 43

[124] Arpita Patra and Ajith Suresh. BLAZE: blazing fast privacy-preserving machine learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020. 4, 8, 9, 13, 21, 22, 37

[125] Chris Peikert. On error correction in the exponent. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 167–183. Springer, Heidelberg, March 2006. doi: 10.1007/11681878_9. 114

[126] David Pointcheval and Olivier Sanders. Short randomizable signatures. In Kazue Sako, editor, *CT-RSA 2016*, volume 9610 of *LNCS*, pages 111–126. Springer, Heidelberg, February / March 2016. doi: 10.1007/978-3-319-29485-8_7. 14, 79, 87

[127] Rahul Rachuri and Ajith Suresh. Trident: Efficient 4pc framework for privacy preserving machine learning. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020. 9

[128] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, volume 9908 of *Lecture Notes in Computer Science*, pages 525–542. Springer, 2016. doi: 10.1007/978-3-319-46493-0_32. URL https://doi.org/10.1007/978-3-319-46493-0_32. 48

[129] Microsoft Research. CrypTFlow: An end-to-end system for secure TensorFlow inference, 2020. https://github.com/mpc-msri/EzPC. 67

[130] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *ASIACCS 18*, pages 707–721. ACM Press, April 2018. 42, 103

[131] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. XONN: XNOR-based oblivious deep neural network inference. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019*, pages 1501–1518. USENIX Association, August 2019. 4, 42, 43, 47, 49, 104

[132] Bita Darvish Rouhani, M. Sadegh Riazi, and Farinaz Koushanfar. Deepsecure: scalable provably-secure deep learning. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 2:1–2:6. ACM, 2018. doi: 10.1145/3195970.3196023. URL https://doi.org/10.1145/3195970.3196023. 42, 104

[133] Amartya Sanyal, Matt J Kusner, Adrià Gascón, and Varun Kanade. Tapas: Tricks to accelerate (encrypted) prediction as a service. In *International Conference on Machine Learning*, pages 4490–4499, 2018. 47

[134] David A. Schultz, Barbara Liskov, and Moses Liskov. Mobile proactive secret sharing. In Rida A. Bazzi and Boaz Patt-Shamir, editors, *27th ACM PODC*, page 458. ACM, August 2008. doi: 10.1145/1400751.1400856. 72

[135] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979. 7, 70

[136] Li Shen, Laurie R Margolies, Joseph H Rothstein, Eugene Fluder, Russell McBride, and Weiva Sieh. Deep learning to improve breast cancer detection on screening mammography. *Scientific reports*, 9(1):1–12, 2019. 3

[137] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016. 3, 42

[138] Nigel P. Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol. In Martin Albrecht, editor, *17th IMA International Conference on Cryptography and Coding*, volume 11929 of *LNCS*, pages 342–366. Springer, Heidelberg, December 2019. doi: 10.1007/978-3-030-35199-1_17. 70, 71, 72, 75

[139] Chuan Zhang Tang and Hon Keung Kwan. Multilayer feedforward neural networks with single powers-of-two weights. *IEEE Transactions on Signal Processing*, 41(8):2724–2727, 1993. 48

[140] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction APIs. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 601–618. USENIX Association, August 2016. 7

[141] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019. 3

[142] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, July 2019. doi: 10.2478/popets-2019-0035. 4, 8, 9, 20, 42, 47, 67, 104

[143] Beta Writer. Lithium-ion batteries. *A Machine-Generated Summary of Current Research. Cham: Springer International Publishing*, 2019. 3

[144] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime G. Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019. URL `http://arxiv.org/abs/1906.08237`. 42

[145] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982. doi: 10.1109/SFCS.1982.38. 4

[146] Yihua Zhang, Marina Blanton, and Fattaneh Bayatbabolghani. Enforcing input correctness via certification in garbled circuit evaluation. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *ESORICS 2017, Part II*, volume 10493 of *LNCS*, pages 552–569. Springer, Heidelberg, September 2017. doi: 10.1007/978-3-319-66399-9_30. 72

[147] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016. 48