Scalable Oblivious Data Analytics

D1.3 Special Purpose MPC Protocols

Frank Blom(TUE), Niek J. Bouman(TUE), Anders Dalskov(AU), Tore Kasper Frederiksen(ALX), Claudio Orlandi(AU), Berry Schoenmakers(TUE), Mark Simkin(AU), Nikolaj Volgushev(ALX), Niels de Vreede(TUE)

The project SODA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583.

Project Information

Scalable Oblivious Data Analytics

Project number:	731583
Strategic objective:	H2020-ICT-2016-1
Starting date:	2017-01-01
Ending date:	2019-12-31
Website:	https://www.soda-project.eu/

Document Information

Title: D1.3 Special Purpose MPC Protocols

ID:	D1.3	Type:	R	Dissemination level:	PU
		Month:	M33	Release date:	30.09.2019

Contributors, Editor & Reviewer Information

Contributors (person/partner)	Frank Blom(TUE): Chapter 4 Niek J. Bouman(TUE): Chapter 4 Anders Dalskov(AU): Chapter 3 Tore Kasper Frederiksen(ALX): Chapter 1 Claudio Orlandi(AU): Chapter 2 Berry Schoenmakers(TUE): Chapter 4 Peter Scholl Mark Simkin(AU): Chapter 2 Nikolaj Volgushev(ALX): Chapter 1 Niels de Vreede(TUE): Chapter 4
Editor (person/partner)	Peter Scholl (AU)
Reviewer (person/partner)	Niek J. Bouman (TUE), Paul Koster (PHI)

Release History

Release	Date issued	Release description / changes made
1.0	30 September, 2019	First release to EU

SODA Consortium

Full Name	**Abbreviated Name**	**Country**
Philips Electronics Nederland B.V.	PHI	Netherlands
Alexandra Institute	ALX	Denmark
Aarhus University	AU	Denmark
Göttingen University	GU	Germany
Eindhoven University of Technology	TUE	Netherlands

Table 1: Consortium Members

Executive Summary

This deliverable presents research carried out in the second half of the SODA project, as part of Work Package 1. The overall goal of Work Package 1 is to improve the state-of-the-art in cryptographic protocols for private big data analytics. This is accomplished in this deliverable by designing and improving special-purpose protocols for dedicated secure computation tasks related to data analytics, and building upon and extending previous work in the SODA project by WP1 and WP2 members.

Firstly, we present implementations and optimizations of multi-party computation (MPC) protocols over rings that were theoretically constructed and reported on in the previous deliverable, and study some specialized tasks which demonstrate their advantages over previous protocols. We also further study the concrete application of privacy-preserving machine learning by presenting efficient protocols for neural network evaluation and ridge regression, and report on their performance when scaling to large networks or large datasets with millions of records. Finally, we look at a new approach to carrying out the preprocessing phase of MPC protocols, which drastically reduces communication for the specific task of preprocessing a large number of oblivious transfers.

Applications of MPC Over Rings for Dishonest Majority. This chapter builds upon SPD$\mathbb{Z}_{2^k}$, which is a new protocol for actively secure MPC against a dishonest majority over rings, developed previously as part of SODA. In this work we demonstrate the advantages that working over rings, instead of fields, can bring to applications of secure computation. We do this by implementing and optimizing SPD$\mathbb{Z}_{2^k}$, and designing and implementing efficient protocols for the specific tasks of equality test, comparison, and truncation over rings. We further show that these operations be be applied to privacy-preserving machine learning, and indeed significantly outperform their field-based competitors. In particular, we implement and benchmark oblivious algorithms for decision tree and support vector machine (SVM) evaluation.

Improved MPC Over Rings for Honest Majority. This chapter presents better protocols for the important setting of actively secure three-party computation over rings with an honest majority, with native support for arithmetic on 32- and 64-bit words. Our starting point is the novel compiler of Damgård et al. by SODA researchers from CRYPTO 2018, and we improve it in several ways: First, we present an improved version of the compiler which reduces the online communication complexity by a factor of 2. Next, we replace their preprocessing protocol (which performs arithmetic modulo a large prime) with a more efficient preprocessing which only performs arithmetic modulo powers of two (and is therefore more efficient). And finally, we present a novel protocol which replaces the preprocessing phase with a "postprocessing" check. The protocols we construct offer different efficiency tradeoffs and can therefore outperform each other in different deployment settings.

We also provide a fully-fledged implementation of our protocols, and extensive benchmarks showing their scalability. Concretely, we achieve a throughput of 3 million 64-bit multiplications per second with each of the three parties located on a different continent and 10 million in the same location, thus achieving the most efficient implementation of a three-party computation protocol for arithmetic circuits modulo 2^{64} with active security.

Secure Evaluation of Quantized Neural Networks. Convolutional neural networks (CNNs) lie at the heart of many data analytics applications like image classification and speech recognition. The need for evaluating such models whilst preserving the privacy of the input provided increases as the models are used for more information-sensitive tasks like DNA analysis or facial recognition.

Research on evaluating CNNs securely has been very active during the last couple of years, with frameworks like SecureNN [128], which can perform evaluation of some CNNs with a multplicative overhead of only 17–33 with respect to evaluation in the clear.

We contribute to this line of research by introducing a technique from the machine learning domain, namely quantization, which allows us to scale secure evaluation of CNNs to much larger networks without the accuracy loss that could happen by adapting the network to the MPC setting. Quantization is motivated by the deployment of ML models in resource-constrained devices, and we show it to be useful in the MPC setting as well. Our results show that it is possible to evaluate realistic models—specifically Google's MobileNets line of models for image recognition—within seconds.

Our performance gain can be mainly attributed to two key ingredients: One is the use of the three-party MPC protocol based on replicated secret sharing, whose multiplication only requires sending one number per party. Moreover, it allows to evaluate arbitrary long dot products at the same communication cost of a single multiplication, which facilitates matrix multiplications considerably. The second main ingredient is the use of arithmetic modulo 2^{64} , for which we develop a set of primitives of indepedent interest that are necessary for the quantization like comparison and truncation by a secret shift.

Efficient Secure Ridge Regression from Randomized Gaussian Elimination. In this chapter we present a practical protocol for secure ridge regression. We develop the necessary secure linear algebra tools, using only basic arithmetic over prime fields. In particular, we will show how to solve linear systems of equations and compute matrix inverses efficiently, using appropriate secure random self-reductions of these problems. The distinguishing feature of our approach is that the use of secure fixed-point arithmetic is avoided entirely, while circumventing the need for rational reconstruction at any stage as well.

We demonstrate the potential of our protocol in a standard setting for information-theoretically secure multiparty computation, tolerating a dishonest minority of passively corrupt parties. Using the MPyC framework, which is based on threshold secret sharing over finite fields, we show how to handle large datasets efficiently, achieving practically the same root-mean-square errors as Scikit-learn. Moreover, we do not assume that any (part) of the datasets is held privately by any of the parties, which makes our protocol much more versatile than existing solutions.

Efficient Secure Computation with Silent Preprocessing. The preprocessing phase of an MPC protocol requires generating some correlated randomness, which typically scales with the size of the function being computed. For very large-scale computations, existing techniques to generate the correlated randomness require a huge amount of communication and storage. A natural tool for addressing these limitations is a *pseudorandom correlation generator* (PCG). A PCG allows two or more parties to securely generate long sources of useful correlated randomness via a local expansion of correlated short seeds and no interaction. PCGs enable MPC with *silent preprocessing*, where a small amount of interaction used for securely sampling the seeds is followed by silent local generation of correlated pseudorandomness.

We present the first concretely efficient PCG for oblivious transfer (OT) correlations, which can be plugged in to reduce the overall communication costs of higher-level protocols such as secret-sharing based MPC and private set intersection. The security of our construction is based on a variant of the learning parity with noise assumption and any correlation-robust hash function. We also implemented, optimized, and benchmarked our actively secure silent OT extension protocol, demonstrating that it offers a more attractive alternative to the OT extension protocol of Ishai et al. [85] in many realistic settings.

About this Document

Role of the Deliverable

The purpose of this deliverable is to study cryptographic protocols for specific tasks, which can be applied to improve the practicality of secure computation for performing analytics on large volumes of data. The deliverable contains several research results on specific applications related to privacy-preserving machine learning, as well as further study, optimization and implementation of protocols developed in previous stages of SODA.

Relationship to Other SODA Deliverables

This deliverable follows on from D1.2 and D2.2, by exploring further the protocols and applications which were developed for those tasks. In particular, Chapters 1–2 build upon protocols for secure computation over rings which we developed in D1.2; Chapters 3 and 4 relate to data analytics applications we first studied in D2.2; and Chapter 5 concerns secure computation in the preprocessing model, which we studied in D1.2. Compared with D1.2, this deliverable lies more on the practical side, with a focus on optimizations and implementations instead of theoretical results.

Structure of this Document

We begin by continuing the study of secure computation over rings, which we began in deliverable D1.2. In Chapter 1, we study the SPD$\mathbb{Z}_{2^k}$ protocol from the perspective of applications, and show how to exploit the fact we are working mod 2^k to improve several important building blocks and applications of MPC. Chapter 2 looks at our compiler for actively secure MPC over rings from D1.2; we optimize this for the special case of 3-party computation and present implementation results, showing it performs favourably to the state-of-the-art. In Chapter 3, we look at the specific application of secure evaluation of neural networks, and present general techniques for optimizing this in MPC, whilst only relying on off-the-shelf trained models. Chapter 4 shows how to securely *train* a ridge regression model in MPC, using randomized Gaussian elimination. We evaluated the performance of our solution by implementing the secure training protocol and testing it on datasets with up to 4 million records. Lastly, in Chapter 5, we look at MPC with *silent preprocessing*, which is a new approach to carrying out the preprocessing phase of MPC protocols with much less communication. For the specific case of preprocessing oblivious transfers, we present a highly efficient construction with active security, and present implementation results showing it outperforms previous approaches in practice.

Table of Contents

Chapter 1

Applications of MPC Over Rings for Dishonest Majority

This chapter is based on the paper *New Primitives for Actively-Secure MPC Over Rings with Applications to Private Machine Learning* co-authored by SODA researchers, published at IEEE Security & Privacy 2019 [50].

1.1 Introduction

1.1.1 Computational Models in Multi-Party Computation

Different multi-party computation (MPC) protocols may require different representations of the function being computed, which can greatly affect the overhead of the protocol, compared with computing f in the clear. The most common approach is to consider the function f as a circuit where input, output, and internal values are from some algebraic structure and gates represent operations over this structure. A typical choice of algebraic structure is the finite field $\mathbf{F}_2$ [78, 18, 109, 92], which means that f computes over bits, addition is equivalent to XOR, and multiplication is equivalent to AND. Another popular choice is the ring $\mathbb{Z}_q$ [20, 56, 95, 61, 44] where addition and multiplication are carried out over the integers modulo some large q. Some protocols also use a binary extension field $\mathbf{F}_{2^k}$ (for a large k), where addition is equivalent to XOR but multiplication is binary polynomial multiplication, which is particularly well-suited to computing certain cryptographic functions such as AES [53, 57, 93].

Each of these have their strengths and weaknesses, for example $\mathbf{F}_2$ is best for bitwise computations such as comparison of two integers, symmetric encryptions and hash functions, while arithmetic modulo q is suitable for arithmetic operations such as computing statistics or linear programming [49]. However, in an application it will often be useful to convert between different representations, depending on the requirements at various stages of the program. For example, this has been done successfully in the ABY framework [58] and subsequent works [105], which convert between arithmetic and binary sharings for applications such as private biometric matching and classification using support vector machines, linear/logistic regression and neural networks. The downside of these approaches is that they only offer security against a passive adversary, or, in the case of [105], can only achieve active security in the restricted setting of three parties with an honest majority (that is, no collusions). MPC protocols with active security against a *dishonest majority* tend to be much more complex, and also typically only support arithmetic modulo q, where q is a large prime. This restriction makes it much more difficult to convert between $\mathbb{Z}_q$ sharings and binary sharings, making the protocols less suitable for applications where these conversions are needed.

1.1.2 The SPD$\mathbb{Z}_{2^k}$ Protocol

Recent work by SODA researchers [47] took a first step in overcoming the above hurdle, with the SPD$\mathbb{Z}_{2^k}$ protocol (named after the SPDZ family of protocols [56, 54]) for actively secure, dishonest majority MPC over $\mathbb{Z}_q$ even when q is not a prime, for example $q = 2^k$. This gives hope that we may be able to exploit arithmetic in $\mathbb{Z}_{2^k}$ to improve the efficiency of applications.

One of the main advantages of working modulo 2^k is that it corresponds naturally to 32/64-bit computations done in standard CPUs, allowing for very simple and efficient implementations without finite field arithmetic. Furthermore, the fact that 32 and 64-bit computation has been the norm for many years means that there are many algorithms optimized for this domain. These cannot trivially be leveraged in MPC applications working over $\mathbf{F}_p$.

Despite these advantages, we note that our previous work [47] only described how to do additions and multiplications securely over $\mathbb{Z}_{2^k}$, which on its own is not enough to realize complex applications. This is because a large number of applications require efficient sub-routines for operations such as equality testing, comparison, and truncation, which do not give rise to efficient arithmetic circuits. Subprotocols for these tasks are well-studied when the computation is over $\mathbf{F}_p$ [51, 111, 40], but it is not immediately clear whether these techniques apply directly to the ring setting over $\mathbb{Z}_{2^k}$. In particular, many of the techniques rely on properties of fields, like the simple fact that division by 2 is possible (as long as the characteristic of the field is not 2). However, this does not work modulo 2^k since 2 is not invertible, so some workarounds are needed.

1.1.3 Contributions

In this work we present new primitives and applications for actively secure computation with a dishonest majority using arithmetic modulo 2^k. We first describe efficient protocols for conversion between binary and arithmetic sharings in $\mathbb{Z}_{2^k}$, and then leverage these to design efficient protocols for equality testing, comparison and truncation that work over the ring of integers modulo 2^k. Finally, we show how these protocols can be applied to solve problems in machine learning, namely, private classification using decision trees and support vector machines (SVMs).

We introduce several optimizations and implement our protocols in the FRESCO framework [6], along with the underlying SPD$\mathbb{Z}_{2^k}$ protocol of Cramer *et al.* [47]. We benchmark and compare our implementation with SPDZ, the state-of-the-art MPC protocol in the dishonest majority setting, also implemented in the FRESCO framework. Our implementation shows a speedup of 4–6x for SPD$\mathbb{Z}_{2^k}$ over SPDZ for computing multiplication, equality and comparison. We also implemented the preprocessing of SPD$\mathbb{Z}_{2^k}$, which is independent of the function to be computed, on top of Bristol-SPDZ [127]. We show this implementation to be highly competitive with the OT-based MASCOT [95] protocol in both WAN and LAN settings. Compared with the more recent Overdrive protocol [96] based on homomorphic encryption, our preprocessing comes close to meeting Overdrive's performance in a LAN setting, but is several times slower in a WAN due to the high communication costs.

To demonstrate our new building blocks, we consider the application of oblivious evaluation of decision trees and SVMs, and show that using our subprotocols for comparison, coupled with the SPD$\mathbb{Z}_{2^k}$ protocol, is around 2–5.3x faster in the online execution phase.

1.1.4 Overview of our Techniques

Both SPDZ, SPD$\mathbb{Z}_{2^k}$, and in fact many contemporary MPC protocols are cast in the *online/offline* setting. In this setting a “slow”, function independent, *preprocessing* phase is first carried out to construct some raw material. When the parties know the specific function to compute, along with their respective inputs, then this raw material is used in the *online* phase to complete the actual computation. The raw material consists of random elements, and random triples for multiplications. During the online phase the random elements can be used to obliviously give input and similarly the multiplication triples can be used to realize multiplication gates. We embrace this model in our protocols, which are typically based on random preprocessed triples, bits or random values, and we also show how to generate this preprocessing data over the appropriate ring for our binary and arithmetic protocols where this was not previously studied.

For our arithmetic-to-binary conversions, we start with the observation that an arithmetic SPD$\mathbb{Z}_{2^k}$ sharing of $x \in \mathbb{Z}_{2^k}$, denoted $[x]$, can be *locally converted* into a sharing of x mod 2, but under a different secret-sharing scheme, namely a SPD$\mathbb{Z}_{2^k}$ instance with $k = 1$. We therefore define this instance with $k = 1$ to be our secret-shared representation of binary values. This also immediately gives us a complete arithmetic-to-binary conversion, assuming we can first bit-decompose x into SPD$\mathbb{Z}_{2^k}$ shares of its bits x_i, which we turn to later. To convert the other way, from binary to arithmetic sharings, we can take a random SPD$\mathbb{Z}_{2^k}$-shared bit $[r]$, convert r to a binary share, then open $x \oplus r$ and use this to adjust $[r]$ into an arithmetic sharing of x, which can be done as a local computation. We can also perform computations on binary-shared values similarly to operations on SPD$\mathbb{Z}_{2^k}$ sharings, using multiplication triples designed for our $k = 1$ instance of SPD$\mathbb{Z}_{2^k}$ to implement AND gates.

To complete the picture, we need to be able to generate the necessary preprocessed random bits over $\mathbb{Z}_{2^k}$ and random multiplication triples over $\mathbb{Z}_2$ (the case of triples over $\mathbb{Z}_{2^k}$ was shown in [47]). Generating random bits modulo 2^k is not as simple as applying standard techniques from the field setting [54] since this relies on taking square roots modulo p, but square roots modulo a power of 2 have a more complex structure, so this cannot be directly applied. However, we show how to exploit the nature of the secret-sharing scheme in SPD$\mathbb{Z}_{2^k}$ such that it is still possible to generate random bits using one multiplication triple, as in SPDZ.

We also show that *binary* SPD$\mathbb{Z}_{2^k}$ triples, with $k = 1$, can be generated *very efficiently* by exploiting TinyOT-style protocols [109, 130] based on XOR-sharings. To do this, we give a conversion protocol which takes a batch of TinyOT-like XOR sharings and converts them to binary SPD$\mathbb{Z}_{2^k}$ sharings with almost no overhead. Since our conversion protocol guarantees that the new sharings will be of the same value, this means creating the new type of triples costs just the same as in TinyOT. This gives us a huge advantage over using native SPD$\mathbb{Z}_{2^k}$ triples, since TinyOT triples can be generated at over *250 000 triples per second*, more than 10x the throughput of our SPD$\mathbb{Z}_{2^k}$ implementation.

For our other key building blocks like secure comparison, equality and bit decomposition, we adapt existing solutions over finite fields [40] to the ring setting. Since many of these protocols have key sub-components consisting only of bit-wise operations, we can apply our conversion protocols to optimize them. We thus obtain very fast online phases for secure comparison and equality, with an online communication complexity of just $O(k)$ bits for k-bit integers. In concrete terms, this gives up to an *85-fold reduction* compared with the online complexity of protocols used in SPDZ, which require sending $O(k)$ field elements per comparison or equality.

1.1.5 Related Work

Many of our subprotocols' optimizations rely on moving between computation over bits and over $\mathbb{Z}_{2^k}$. Several previous works have studied conversions between different types of secret-sharing representations for MPC, most notably the ABY framework [58], which has passively secure two-party protocols for converting between arithmetic, binary and Yao-based secret data types. Chameleon [121] extended this to a setting with an external, non-colluding third party to assist in the computation, and ABY^3 [105] extended this to a more general three-party honest majority setting, also with some support for active security. On the theoretical side, share conversion between different secret-sharing schemes was first studied by Cramer, Damgård and Ishai [48].

In the last few years there has been a lot of research in private machine learning applications using secure computation. For our applications to decision tree and SVM evaluation, the most relevant are the works by de Cock *et al.* [45], Demmler *et al.* [121] and Makri *et al.* [104]. For a more thorough survey including other machine learning applications, we refer the reader to [106, 105].

On the side of MPC primitives like comparison, there has been some other work in the setting of general, dishonest majority MPC over the ring $\mathbb{Z}_{2^k}$ [102]. Although their protocols are quite efficient asymptotically, they unfortunately have quite large hidden constants and local computation, compared to the state-of-the-art protocols working over fields [40], and in turn our protocols as well.

Even though SPD$\mathbb{Z}_{2^k}$ is the only MPC protocol we are aware of that works over the ring $\mathbb{Z}_{2^k}$ *and* is actively secure against a dishonest majority, other authors have worked on MPC protocols over $\mathbb{Z}_{2^k}$, but with less stringent security requirements. Of particular interest is Sharemind [27], as this scheme also allows mixing boolean and arithmetic operations. However, security is only in the passive, 3-party setting for an honest majority. Sharemind has also been extended to the active case [115]. Another relevant work in this area is the compiler by Damgård *et al.* [55], which can transform a passively secure protocol for t corruptions into an actively secure protocols for $\sqrt{t}$ corruptions (meaning an honest majority). Recently Araki *et al.* [11] presented a highly efficient stand-alone protocol for passive security in the honest majority setting.

1.2 Preliminaries

1.2.1 Notation

Given a natural number M, we denote by $\mathbb{Z}_M$ the set of integers x such that $0 \leq x \leq M-1$. We abbreviate the congruence $x \equiv y \mod 2^k$ as $x \equiv_k y$. We let $x \bmod M$ denote the remainder of x when divided by M, and we take this representative as an element of the set $\mathbb{Z}_M$. When we write $c = a \overset{?}{<} b$, we mean that c is 1 if $a < b$, and 0 otherwise.

1.2.2 Background on SPD$\mathbb{Z}_{2^k}$ Shares and Core Protocols

Our protocols build upon the secret-sharing scheme from SPD$\mathbb{Z}_{2^k}$ [47] based on additive secret-sharing with information-theoretic MACs, and its subprotocols used for computing on shares. The main idea behind this secret-sharing scheme is that, to perform a secure computation on additive shares modulo 2^k with active security, the parties will run a computation *over a larger ring* modulo 2^{k+s}, where $\sigma = s - \log(s)$ is a statistical security parameter, but *correctness is only guaranteed modulo* 2^k. The reason for this is that in a ring with many zero-divisors, traditional information-theoretic MACs cannot protect the integrity of an entire ring element $x' \in \mathbb{Z}_{2^{k+s}}$, however, they can offer integrity on the lower-order k bits, namely $x = x' \bmod 2^k$.

Given $x \in \mathbb{Z}_{2^k}$, we denote by $[x]_{2^k}$ the situation in which the parties have additive shares $x^1, \ldots, x^n, m^1, \ldots, m^n \in \mathbb{Z}_{2^{k+s}}$ and $\alpha^1, \ldots, \alpha^n \in \mathbb{Z}_{2^s}$ such that $x \equiv_k \sum_j x^j$ and $\left(\sum_j \alpha^j\right) \cdot \left(\sum_j x^j\right) \equiv_{k+s} m^j$. If there is no chance of ambiguity we use $[x]$ to denote $[x]_{2^k}$ when k is a large integer, e.g. $k = 32$ or 64.

We now summarize the core protocols for manipulating SPD$\mathbb{Z}_{2^k}$ shares, based on [47], which we use.

Input value. $[x] \leftarrow \mathsf{Input}(x, P_i)$, where $x \in \mathbb{Z}_{2^k}$. Secret-shares and authenticates a private input x from party P_i.

Linear operations. $[z] \leftarrow a[x] + [y] + b$. Any linear function or addition by a constant can be performed without interaction, resulting in a sharing of $z = ax + y + b \mod 2^k$. The shares $z^j, t^j \in \mathbb{Z}_{2^{k+s}}$ of z and its MAC can be computes as follows. Let $x^j, m^j \in \mathbb{Z}_{2^{k+s}}$ be the shares of x and the shares of its MAC for party P_j, and let $y^j, h^j \in \mathbb{Z}_{2^{k+s}}$ be the analogous for y. Party P_1 sets $z^1 = ax^1 + y^1 + b \bmod 2^{k+s}$ and, for $j \geq 2$, party P_j sets $z^j = ax^j + y^j \bmod 2^{k+s}$. Finally, all parties P_j compute $t^j = am^j + h^j + b\alpha^j \bmod 2^{k+s}$.

Secret-shared multiplication. $[z] \leftarrow [x] \cdot [y]$. Given a secret-shared multiplication triple, that is, shares $[a], [b], [c]$ for random $a, b \in \mathbb{Z}_{2^k}$ and $c = a \cdot b \bmod 2^k$, a sharing of the product of any two sharings $[x]$ and $[y]$ can be obtained with 1 round of interaction.

Open. $x' \leftarrow \mathsf{Open}_{k'}([x], P_i)$. Opens the sharing $[x]$ modulo $2^{k'}$ towards party P_i, where $k' \leq k$, so that P_i learns only $x' := x \bmod 2^{k'}$. The MAC on $[x]$ is checked for authenticity, although sometimes when opening many values at once, the checks can be deferred and batched for greater efficiency. If k' is omitted, we assume $k' = k$. Furthermore, if the argument P_i is omitted, we assume the share is opened towards all parties.

Security Model

The security properties of the above protocols, and all the protocols in this work, can be formalized using the *arithmetic black box model*, see for instance [102]. In this exposition we omit the formal definitions and proofs in this model. In the full version we include proofs of basic correctness and privacy properties

1.2.3 Preprocessing Material in SPD$\mathbb{Z}_{2^k}$

The SPD$\mathbb{Z}_{2^k}$ protocol runs in two separate phases, the *preprocessing phase*, which is independent of the parties' inputs and can be done in advance, and the *online phase*. There are several different types of random preprocessing data that are needed for different operations in the online phase. As mentioned above, we need a preprocessed multiplication triple for every secret-shared multiplication. For each input by a party P_i, we also need a preprocessed random shared mask known to P_i. Additionally, in some of our protocols we use random shared bits, which we show how to generate from a multiplication triple. The Open protocol also uses a preprocessed random mask, however, when opening and checking MACs on many values in a batch the same mask can be used for one check of all values, so we do not count this cost in our evaluation.

Multiplication triples are the most performance-intensive type of preprocessing data to generate. Random bits cost around the same as a triple; together these form the bottleneck of the preprocessing phase. The masks used for inputs and opening are cheaper, requiring around 30x less communication than triples when using the protocols from [47].

1.3 Converting Between Binary and Arithmetic Sharings

1.3.1 Binary sharings

To represent a binary shared value, we simply use a standard mod 2^k sharing with $k = 1$. That is, the bit b and the MAC $\alpha \cdot b$ are both additively shared modulo 2^{s+1}, where the shares of b are only guaranteed to be of the correct value modulo 2. We denote this by $[b]_2$, in contrast with $[b]$ for an arithmetic sharing. Given two binary shared values $[a]_2$ and $[b]_2$, if the parties locally add the shares then they obtain a valid sharing of the XOR of the two bits, $a \oplus b$. Multiplication corresponds to AND, and requires a binary shared triple $[x]_2, [y]_2, [z]_2$ such that $z \equiv x \cdot y \mod 2$. We remark that, just as with SPD$\mathbb{Z}_{2^k}$ triples, it is *not* necessary for the multiplicative relation to hold modulo 2^{s+1}. So, even though the parties hold additive shares of x, y and z modulo 2^{s+1}, we may have $z \neq x \cdot y \mod 2^{s+1}$. In fact, this is exploited by our protocol for efficiently converting XOR-shared binary triples into $[\cdot]_2$-sharings.

1.3.2 Efficient binary triple generation

The online phase of the SPD$\mathbb{Z}_{2^k}$ protocol works for any k, but unfortunately its offline phase (more specifically, the sacrifice step in the triple generation protocol) requires k to be at least the security parameter. To combine SPD$\mathbb{Z}_{2^k}$ with binary operations, we need another way of generating multiplication triples. One way could be to generate triples with a large k and then reduce them to get $[\cdot]_2$ shares (as explained in Sec. 1.3.4). However, binary triples $[x]_2, [y]_2, [z]_2$ can be generated *much more efficiently* by exploiting TinyOT-style protocols [109, 129, 130], which generate triples with *XOR-shared* MACs and shares, as we now show.

We will present a general technique for converting between two different types of sharings, which both support linear computations over $\mathbf{F}_2$. Given this conversion protocol, we can convert triples generated using TinyOT—or any other authenticated, $\mathbf{F}_2$-linear secret-sharing scheme—into a triple based on our binary share representation.

Let $\langle x \rangle$ denote that the bit x is shared and authenticated using TinyOT, that is, each party P_i holds a bit x^i and a MAC $M^j_{x_i} \in \{0,1\}^s$ on x^i, as well as a MAC key $K^i_{x_j} \in \{0,1\}^s$ for P_j's share x^j, for all $j \neq i$. Each party also has a global MAC key $\Delta^i \in \{0,1\}^s$. The shares and MACs are set up such that $x = \bigoplus_i x^i$ and $M^j_{x_i} = K^j_{x_i} \oplus x^i \Delta^j$, for all $j \neq i$. TinyOT-shared values can be XORed together locally and multiplied by 0/1 constants in the usual manner. To convert a batch of TinyOT sharings $\langle x_1 \rangle, \ldots, \langle x_m \rangle$ into $[\cdot]_2$ sharings, we use the protocol in Fig. 1.1. The basic idea is that, for each input x, every party will authenticate their XOR shares x^i using SPD$\mathbb{Z}_{2^k}$ to create a new binary sharing and obtain $[x]_2$. Note that even though the original shares $x^i \in \{0,1\}$ are now summed over the integers modulo 2^{s+1} to form $[x]_2$, they should still give a valid sharing of x mod 2, since the upper s bits do not matter. To verify that everyone inputs the correct shares x^i, we take a random $\mathbf{F}_2$-linear combination of all m shares, masked by an additional random share, then open this using both the TinyOT and the SPD$\mathbb{Z}_{2^k}$ sharings and check consistency. This check has soundness $1/2$, so we repeat it σ times (using σ additional random masked bits) to achieve a cheating probability of $2^{-\sigma}$.

We prove the following Lemma in the full paper [50].

Lemma 1. *If the inputs $\langle x_1 \rangle, \ldots, \langle x_m \rangle$ form consistent TinyOT sharings of bits $x_1, \ldots, x_m$ under uniformly random MAC keys, then the output sharings $[x_1]_2, \ldots, [x_m]_2$ form consistent SPD$\mathbb{Z}_{2^k}$ sharings with $k = 1$, except with probability at most $2^{-\sigma}$.*

Protocol $\Pi_{\langle x\rangle \to [x]_2}$

INPUT: TinyOT sharings $\langle x_1\rangle, \dots, \langle x_m\rangle$.
OUTPUT: Binary sharings $[x_1]_2, \dots, [x_m]_2$.

1. Sample s additional random TinyOT-shared bits $\langle r_1\rangle, \dots, \langle r_\sigma\rangle$.
2. Each party P_i inputs the shares $x_1^i, \dots, x_m^i, r_1^i, \dots, r_\sigma^i$ with $\mathsf{Input}(\cdot, P_i)$, and then the parties sum up the shares to obtain (possibly incorrect) sharings $[x_1]_2, \dots, [x_m]_2$ and $[r_1]_2, \dots, [r_\sigma]_2$.
3. Sample $m \cdot s$ random bits $\chi_{i,j} \leftarrow_R \{0,1\}$, for $i = 1, \dots, m$ and $j = 1, \dots, \sigma$, using a coin-tossing protocol.
4. For each j, let $y_j = \mathsf{Open}_2([r_j]_2 \oplus \sum_{i=1}^m \chi_{i,j} \cdot [x_i]_2)$ and $y'_j = \Pi_{\mathsf{TinyOT.Open}}(\langle r_j\rangle_2 \oplus \sum_{i=1}^m \chi_{i,j} \cdot \langle x_i\rangle_2)$.
5. Check that $y_j = y'_j$ for all j. If not, abort.
6. Output the sharings $[x_1]_2, \dots, [x_m]_2$.

Figure 1.1: TinyOT share to binary SPD$\mathbb{Z}_{2^k}$ share conversion. $\Pi_{\mathsf{TinyOT.Open}}$ denotes the TinyOT share opening protocol.

1.3.3 Arithmetic to Binary

Given a SPD$\mathbb{Z}_{2^k}$ sharing $[x]$, the parties can obtain a correct binary sharing of the least significant bit of x by simply truncating the upper $k-1$ bits of the shares and MAC shares of $[x]$. This protocol is given in Fig. 1.2, and it is easy to see that this gives a consistent sharing of x mod 2.

Protocol Π_{A2B}

INPUT: Arithmetic sharing $[x]$.
OUTPUT: Binary sharing $[y]_2$, where $y = x \mod 2$.

1. Let x^i, m_x^i be P_i's share and MAC share of $[x]$.
2. P_i defines $y^i = x^i \bmod 2^{s+1}$ and $m_y^i = m_x^i \bmod 2^{s+1}$ to obtain shares of $[y]_2$.

Figure 1.2: Arithmetic to binary SPD$\mathbb{Z}_{2^k}$ share conversion

1.3.4 Binary to Arithmetic

To convert a binary share $[x]_2$ into a SPD$\mathbb{Z}_{2^k}$ sharing, we use the protocol in Fig. 1.3. This uses a subprotocol Π_{RandBit} for generating a sharing $[r]_{2^k}$ of a random bit r known to none of the parties, which we show how to do in the full paper [50]. Given this, we can locally compute $[r]_2$ using arithmetic-to-binary conversion, and then open $c = x + r \mod 2$, which perfectly hides b. Finally, using c and $[r]$ we can locally compute an arithmetic sharing of $x = c \oplus r$.

1.4 Applications

A major application of our new conversion protocols is to perform more efficient bitwise operations like secure comparison and truncation; in the full paper [50] we show how our protocols can be applied to these building blocks. In this section, we demonstrate some concrete use-cases which take advantage of these protocols.

Protocol Π_{B2A}

INPUT: Binary sharing $[x]_2$.
OUTPUT: Arithmetic sharing $[x]$.

1. Let $[r] = \Pi_{\mathsf{RandBit}}()$.
2. Compute $[r]_2 = \Pi_{\mathsf{A2B}}([r])$.
3. Let $c = \mathsf{Open}_2([x]_2 + [r]_2)$.
4. Output $[x] = c + [r] - 2 \cdot c \cdot [r]$.

Figure 1.3: Binary to arithmetic SPD$\mathbb{Z}_{2^k}$ share conversion

1.4.1 Decision Trees

We consider the machine-learning application of decision trees which is used for classification. A *decision tree* is a function $\mathsf{T} : \mathbb{R}^n \to \mathbb{Z}_q$, where n is called the dimension of the *feature space* and q is the amount of possible output categories. The input $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ to T is called the *feature vector*. The function T is implemented as a binary tree with m internal nodes, where each internal node v_j for $j \in [1,m]$ has associated a Boolean function $f_j : \mathbb{R}^n \to \{0,1\}$ s.t. $f_j(\boldsymbol{x}) = x_{\iota_j} \overset{?}{<} t_j$ where $\iota_j \in \mathbb{Z}_n$ is an index into the feature vector $\boldsymbol{x}$ and $t_j \in \mathbb{R}$ is a threshold. Thus $f_j(\boldsymbol{x})$ evaluates to 1 if and only if $x_{\iota_j} \leq t_j$, and 0 otherwise. Each leaf node of the tree is associated with an output value $z \in \mathbb{Z}_q$. Now to evaluate $\mathsf{T}(\boldsymbol{x}) = z$, start at the root node and evaluate $f_1(\boldsymbol{x})$. If $f_1(\boldsymbol{x}) = 0$ then proceed to evaluate the left child, if instead $f_1(\boldsymbol{x}) = 1$ then proceed to evaluate the right child. Continue in this manner until reaching a leaf and return the value z of this leaf.

For simplicity, and since we want to hide the structure of the tree, we assume that it is complete. We note that this is always possible as dummy nodes can be inserted as needed, which always evaluate to 0.

We index nodes starting with 1 for the root node and then indexing by reading each layer top to bottom and left to right; thus if v_j is an internal node then v_{2j} is the left child of v_j and v_{2j+1} is the right child. We say the depth is the amount of nodes in the path from the root to, and including, the leaf; defining the root to be level 0. Thus the tree will have $m = 2^{d-1} - 1$ internal nodes and 2^d leaves. Note that the leaves will have index 2^d to 2^{d+1}.

Concretely we define T as a tuple of values $(\boldsymbol{t}, \boldsymbol{v}, \boldsymbol{z})$, where $\boldsymbol{t} \in \mathbb{R}^m$, $\boldsymbol{v} \in \mathbb{Z}_n^m$ and $\boldsymbol{z} \in \mathbb{Z}_q^{2^d}$. That is, $\boldsymbol{t} = (t_1, \ldots, t_m)$ and $\boldsymbol{v} = (v_1, \ldots, v_m)$ are lists of cardinality m. We view as ordered such that the j'th entry describe the j'th internal node in the tree. That is, each internal node v_j will compute the value $f_j = x_{v_j} \overset{?}{<} t_j$. $\boldsymbol{z} = (z_1, \ldots, v_{2^d})$ is an ordered list of integers, each representing an output of a leaf, thus each leaf node v_j (i.e. with $j \in]m, m+2^d]$) will output the value $f_j = \boldsymbol{z}_{j-2^d}$.

Furthermore we consider the two-party setting where one party, called the *client* holds the feature vector $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$. The other party, called the *server* holds the decision tree T. The parties then wish to compute $\mathsf{T}(\boldsymbol{x}) = z$ where the client learns z and the server learns nothing.

To evaluate a decision tree privately we work over a finite set of integers $\mathbb{Z}_{2^k}$ instead of the real numbers. We convert a model based on real numbers by simply multiplying every decimal number in the model by a set constant and then rounding to nearest integer. This of course causes loss in accuracy, however, this rarely causes a problem and for real data the constant does not necessarily have to be large to avoid losing classification accuracy [104]. We furthermore note that this conversion still allows us to work with negative integers by considering the positive integers up to 2^k as a value in two's complement, thus representing the positive integers up to $2^{k-1}-1$ and following these, the negative integers from -2^{k-1} to -1. Because our computations will take place over a ring this representation will ensure arithmetic operations act as expected (assuming no over- and underflow).

An actively secure protocol

Our protocol takes departure in the work by De Cock *et al.* [45] which presents a protocol for evaluating decision trees based on secret sharing. We picked this protocol since it works in the arithmetic black box setting, whereas other approaches such as the one by Wu *et al.* [132] or Joye and Sahali [90] require homomorphic encryption. Still, the scheme by De Cock *et al.* is only secure in the semi-honest setting. We show how to make it actively secure by adding a cheap extra step.

The overall idea of their scheme is to first pick each relevant value from the input feature vector $\boldsymbol{x}$ for each node j, i.e. x_{v_j}. This is done by having the party holding the tree, P_1, input an n-bit vector for each of the m nodes. This bitvector will contain a single 1-bit in the position of the feature to use. That is, we associate a bit $c_{j,i} \in \{0,1\}$ with each feature for each node (i.e. for all $i \in [1,n], j \in [1,m]$) s.t. $\sum_{i\in[1,n]} c_{j,i} = 1$ and $c_{j,v_j} = 1$. With these indicator bits we can arithmetically compute the attribute to use in the j'th node as $\sum_{i\in[1,n]} c_{j,i} \cdot x_i$.

If the tree holder is actively corrupted then it will be able to input value $c_{j,i}$ s.t. $\sum_{i\in[1,n]} c_{j,i} \neq 1$. This is a problem since this would allow a linear combination of $(x_1,\ldots,x_n)$ to be used for the comparison in each node of the tree. This would make it hard to write a simulation proof since the simulator would not know $\boldsymbol{x}$. To fix this issue we propose a solution that consists of enforcing that $c_{j,i}$ is a bit, then open $\sum_{i\in[1,n]} c_{j,i}$ for $j \in [1,m]$ and check if this is always 1. It is easy to see that this check is sufficient and clearly does not leak any information (as it is public knowledge that the opened value is supposed to be 1). Furthermore, it is also easy to enforce that $c_{j,i}$ is a bit, even if the whole ring $\mathbb{Z}_{2^k}$ is allowed as input: simply compute and open the value $(1-c_{j,i}) \cdot c_{j,i}$ and check if it is 0. Again it can be argued that this is sufficient as $c_{j,i}$ equal to 0 or 1 are the only values for which $(1-c_{j,i}) \cdot c_{j,i} = 0$ when working over $\mathbb{Z}_{2^k}$.

Adding this check allows us to compute the correct attributes for each node with active security. Via the attributes the output of the comparison in each node can be computed by the comparison subprotocol; the output is a bit indicating whether to go left (0) or right (1) down the tree. To evaluate the tree obliviously it is not possible to simply follow the correct path from the root to a leaf, as this would leak too much. Thus, we must visit every node in the evaluation. This is done by computing a bit for each leaf, which is the product of the output of the comparison for all the nodes on the path to the root.[1] There will be only one leaf for which this bit is 1. This is the leaf whose value is the final output of the decision tree evaluation. Since the evaluator is oblivious to which leaf this is, we multiply the bit of each leaf with the leaf's value and sum this for all leaves. Because the bit for every leaf, other than the correct one is 0, the output of this computation gives the correct result. This means that the comparisons can be done once; for each internal node of the tree we can then compute if it is part of the root-to-leaf path that is the result of the decision tree evaluation. Still, this requires $O(d)$ rounds of communication as all nodes on a given layer are dependent on a partial result of the nodes higher up the tree.

We can compute the partial values of all nodes in the tree using a "reduction" approach by exploiting the fact that multiplication is associative, i.e. that $x_1 \cdot x_2 \cdot x_3 \cdot x_4$ can be computed as $(x_1 \cdot x_2) \cdot (x_3 \cdot x_4)$, rather than $((x_1 \cdot x_2) \cdot x_3) \cdot x_4$. Thus we can compute the product of d values with $d-1$ multiplications and $\log(d-1)$ sequential rounds. For each node in every second layer from the root to the leaves, we

[1]The output of the comparison is negated for each node if it is a left child.

compute the product of the output of its comparison with the output of the comparison of its parent (negated if it is a left node). Next we use these results to compute a product for every four layers, by multiplying the result of every node with the result of its grandparent (negated if its parent is a left child). We continue until we have computed a product between every layer in the tree.

Computing these products dominates protocol round cost, since both selecting the feature for all nodes, along with computing the comparison can be done in constant rounds (assuming we use the constant round comparison protocol).

We note that De Cock *et al.* have implemented their protocol using boolean values, whereas we use arithmetic values. Using boolean values and replacing multiplication and addition with component-wise AND and XOR respectively would unfortunately not directly work on our fix to get active security. This is because XOR'ing two 1's would give 0, so an actively corrupted model holder would be able to have the classification happen using XOR combinations of the different values of the inputting party's feature vector. Even more importantly, as the feature values are not binary but rather elements from $\mathbb{Z}_{2^k}$, using a binary protocol would require k multiplications (AND gates) to compute $c_{j,i} \cdot x_i$ for $i \in [m]$ and $j \in [n]$, needed for each node in the tree. Even for relatively small values of k, like 32, this would probably not be faster using a binary protocol. In particular, using the optimized TinyOT protocol [129] this would be slower as the construction of a TinyOT triple is only about 12x faster than a SPD$\mathbb{Z}_{2^k}$ triple.

1.4.2 Support Vector Machines (SVMs)

We consider the machine-learning application of Support Vector Machines (SVMs), which is a type of supervised learning model used for classification. In its simple form it is used as a binary classifier, but it can easily be extended to classify data into any finite set of categories. More specifically an *SVM* is a function $\mathscr{S} : \mathbb{R}^n \to \mathbb{Z}_q$, where n is the dimension of the feature space and q the amount of categories (each represented by a non-negative integer). Similarly to the decision trees, the input $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ to the function $\mathscr{S}$ is called the *feature vector*. The SVM $\mathscr{S}$ is implemented as a matrix $\boldsymbol{F} \in \mathbb{R}^{q \times n}$ where the rows are known as the *support vectors* and a vector $\boldsymbol{b} = (b_1, \ldots, v_q) \in \mathbb{R}^n$ which is called the *bias*. Conceptually, each support vector, along with a scalar from the bias vector, can classify an input $\boldsymbol{x}$ into a specific category (or not). Specifically denoting the rows of $\boldsymbol{F}$ as $F_1, \ldots, F_q$, the value $F_i \cdot \boldsymbol{x} + b_i$ is computed to give a score of how likely $\boldsymbol{x}$ is to be in category i. Thus, to find the most likely category of $\boldsymbol{x}$ we compute category$(\boldsymbol{x}) = \text{argmax}_{i \in [1,q]} F_i \cdot \boldsymbol{x} + b_i$ where the result is an integer representing the corresponding category.

Like the case for decision trees we consider the two-party setting where one party, called the *client* holds the feature vector $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$. The other party, called the *server* holds the SVM $\mathscr{S}$. The parties then wish to compute $\mathscr{S}(\boldsymbol{x}) = z$ where the client learns z and the server learns nothing. Similarly to the decision trees, we work over a finite set of integers $\mathbb{Z}_{2^k}$, assuming two's complement representation to allow for integers in the range $[2^{k-1}, 2^k)$

An actively secure protocol

Our protocol follows the equation for SVM classification, category$(\boldsymbol{x}) = \text{argMax}_{i \in [1,q]} F_i \cdot \boldsymbol{x} + b_i$, very straight forward: In parallel compute the multiplication part of the inner products between $\boldsymbol{x}$ and F_i for all $i \in [1, q]$, as these are all independent. Next we note that addition does not require communication and thus we sequentially have the parties sum up the component-wise product computed, in order to compute the whole inner product. Next, for each inner product the parties add b_i. These steps only require constant rounds of communication and $q \cdot n$ multiplications. Finally computing the largest element of the q element list is done in $O(\log(q))$ rounds as follows: In a recursive manner divide the list of elements in halves until two or three elements remain. Compare these obliviously, and based on this comparison construct a binary list where the index of the maximum of these two or three elements is 1 and the is rest 0. This requires one or two comparisons and at most four multiplications. The merging of the partial results then require $O(q)$ comparisons and multiplications. Thus we end with a total of $O(q \cdot \log(q))$ comparisons and multiplications for the arg-max computation.

1.5 Performance Evaluation

We now evaluate the concrete performance of our implementation of the online phase (Sec. 1.5.1) and the offline phase (Sec. 1.5.2). In the full paper [50], we also describe several optimizations we used and present further analyses.

For the online phase, we run micro-benchmarks for our basic primitives as well as end-to-end evaluations of our two high-level applications on realistic datasets. We then compare our online implementation of SPD$\mathbb{Z}_{2^k}$ in FRESCO with the baseline SPDZ implementation in FRESCO. The SPDZ implementation in FRESCO is based on SPDZ-2 [54], which is the most recent and efficient online protocol for SPDZ. In our evaluation of the offline phase, we evaluate the SPD$\mathbb{Z}_{2^k}$ triple generation protocol across varying security parameters and network configurations. We then compare our offline implementation in the Bristol-SPDZ C++ framework with the two most recent and efficient protocols for SPDZ triple generation; MASCOT [95] and Overdrive [96]. Both of these are also implemented in the Bristol-SPDZ framework, which ensures a more fair comparison. Our implementation forms part of MP-SPDZ [107], a successor to Bristol-SPDZ.

Furthermore, we are unaware of any other *practically competitive* protocols considering a dishonest majority of malicious parties in the arithmetic setting and thus believe that comparing to SPDZ is sufficient.

We chose to benchmark our protocols in the two-party setting, although all our constructions (except the protocols for the specific setting of oblivious decision tree and SVM evaluation) generalize to an arbitrary amount of parties. We did this for simplicity and since both SPDZ and SPD$\mathbb{Z}_{2^k}$ generalize to more parties with similar overheads.

Setup. We run all experiments in the two-party setting. Each party executes on an m5d.xlarge AWS EC2 instance running Ubuntu 16.04, with 4 vCPUs and 16GB memory. The instances are hosted within the same region and connected over an up to 10 Gbps link. To investigate how different network settings affect the performance of our protocols, we use `tc` to simulate bandwidth restrictions and latency. For all experiments, we performed a minimum of 20 total runs and report the average result. We discard the first run in order to ensure the JVM has warmed up.

1.5.1 Online Phase

For our online phase experiments, we consider two bit length settings. For the low bit length setting, we use $k = s = 32$ (total bit length of 64) which supports 32-bit comparisons and equality operations and affords 26 bit statistical security. We compare this setting to running SPDZ over a 64 bit field; the larger field is necessary to ensure at least 26 bits of statistical security in the comparison protocol used by SPDZ. Similarly, we compare the larger bit setting with $k = 64$, $s = 64$, total bit length 128, and 57 bit statistical security to SPDZ over a 128 bit field with 57 bit statistical security.[2]

[2] 26, respectively 57 bits of security, are chosen for a fair comparison with SPD$\mathbb{Z}_{2^k}$, as SPD$\mathbb{Z}_{2^k}$ has a logarithmic deterioration of the statistical security, because of batched MAC checks.

Table 1.1 shows throughput times (operations per second) for three non-linear operations: multiplication, equality, and comparison on a 1 Gbps network. We believe a 1 Gbps LAN to be a suitable setting for the family of SPD$\mathbb{Z}_{2^k}$ and SPDZ protocols; the high latency of lower bandwidth WAN networks would significantly limit performance due to the protocols' non-constant round complexity. Constant round protocols are more appropriate for such settings. Conversely, we do not report numbers for a faster network since at 1 Gbps our implementation is not network-bound.

Table 1.1: Throughput in elements per second for the online phase of micro operations over 1 Gbps network. The factor columns express the runtime improvement factor of SPD$\mathbb{Z}_{2^k}$ over SPDZ in FRESCO.

	$k=32$			$k=64$		
	SPD$\mathbb{Z}_{2^k}$ ($\sigma=26$)	SPDZ ($\sigma=26$)	Factor	SPD$\mathbb{Z}_{2^k}$ ($\sigma=57$)	SPDZ ($\sigma=57$)	Factor
Multiplication	687041	141346	4.9x	522258	114071	4.6x
Equality	15334	3213	4.8x	6902	1282	5.4x
Comparison	9153	1769	5.2x	4514	756	6.0x

We obtain the throughput numbers from batched runs, i.e., parallel[3] operations with batched communication. We use batches of 100,000 parallel operations for multiplications and 5,000 for equality and comparison.

For multiplications we see between a 4.6 and 4.9-fold improvement for the different bit-length settings. This performance gain stems from a speed up in local computation as well as reduced communication. Local computation improves since we do not need to perform modular reductions and use a custom class for ring elements of specific bit-length (64 and 128 bit) which significantly outperforms BigInteger arithmetic. The total amount of data sent is also reduced; for all protocols that require communicating an element to the other parties, we only need to send the k least significant bits, as opposed to an entire element for SPDZ. This alone cuts communication in half.

Comparison and equality (for $k=64$) show an even higher increase in performance, with the biggest improvement for comparison, six-fold for $k=64$ and five-fold for $k=32$.

Switching to boolean mode for the comparison protocol replaces a majority of the underlying multiplications with bit-multiplications, which require sending only 2 bits per party, in contrast to two whole field elements. This drastically reduces communication. The improvement in throughput is not directly proportional to the reduction in communication since our implemention is not network-bound at 1 Gbps. We nonetheless observe an improvement since reducing data sent also reduces the amount of local serialization and data copying FRESCO does as part of networking.

Equality also benefits from switching to boolean mode, though the performance improvement is less pronounced; we operate in arithmetic mode by default and must convert the boolean output of the Π_{EQZ} protocol to an arithmetic sharing. This introduces an additional protocol round, but we avoid this conversion for comparisons.

We note that for $k=32$, multiplication yields a slightly higher relative improvement than equality. This is due to the fact that the benefit of reduced communication for equality is not high enough to outweigh the internal framework-related overhead of executing a more complex protocol.

The lower communication of multiplication and comparison directly affects the communication and computation required for the more advanced applications of decision trees and SVMs, as can be seen in Tab. 1.2 and 1.3.

[3]Parallel here does not imply running on multiple threads; it merely means that the operations are independent and communication can thus be batched.

Table 1.2: Online phase benchmarking of evaluation of decision trees over 1 Gbps network. The factor columns express the runtime improvement factor of SPD$\mathbb{Z}_{2^k}$ over SPDZ in FRESCO. Times are in milliseconds per sample.

			$k = 32, \sigma = 26$			$k = 64, \sigma = 57$		
Dataset	Depth, Num. Features	Batch Size	SPD$\mathbb{Z}_{2^k}$	SPDZ	Factor	SPD$\mathbb{Z}_{2^k}$	SPDZ	Factor
Hill Valley	3, 100	1	21 ms	24 ms	1.2x	26 ms	34 ms	1.3x
Spambase	6, 57	1	48 ms	104 ms	2.2x	56 ms	128 ms	2.3x
Diabetes	9, 8	1	80 ms	215 ms	2.7x	122 ms	443 ms	3.6x
Hill Valley	3, 100	5	6 ms	10 ms	1.7x	7 ms	15 ms	2.1x
Spambase	6, 57	5	14 ms	40 ms	2.9x	17 ms	68 ms	4.0x
Diabetes	9, 8	5	41 ms	185 ms	4.5x	78 ms	376 ms	4.8x

Table 1.3: Online phase benchmarking of SVM evaluation over 1 Gbps network. The factor columns express the runtime improvement factor of SPD$\mathbb{Z}_{2^k}$ over SPDZ in FRESCO. Times are in milliseconds per sample.

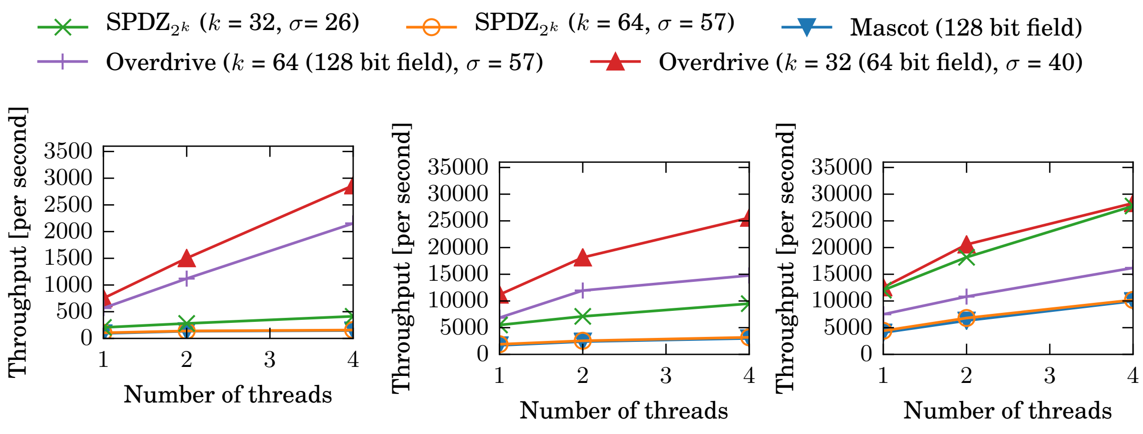
			$k = 32, \sigma = 26$			$k = 64, \sigma = 57$		
Dataset	Num. Classes, Features	Batch Size	SPD$\mathbb{Z}_{2^k}$	SPDZ	Factor	SPD$\mathbb{Z}_{2^k}$	SPDZ	Factor
CIFAR	10, 2048	1	82 ms	214 ms	2.6x	99 ms	255 ms	2.6x
MIT	67, 2048	1	379 ms	1318 ms	3.5x	499 ms	1582 ms	3.2x
ALOI	463, 128	1	242 ms	857 ms	3.5x	362 ms	1312 ms	3.6x
CIFAR	10, 2048	5	39 ms	168 ms	4.3x	57 ms	209 ms	3.7x
MIT	67, 2048	5	225 ms	1101 ms	4.9x	294 ms	1428 ms	4.9x
ALOI	463, 128	5	162 ms	741 ms	4.6x	244 ms	1220 ms	5.0x

1.5.2 Offline Phase

Fig. 1.4 compares our implementation of triple generation to the two state-of-the-art preprocessing protocols of the SPDZ family; MASCOT [95], and Overdrive [96]. All three implementations are part of the MP-SPDZ framework [107]. We first note that SPD$\mathbb{Z}_{2^k}$ saturates the network for all number threads we tested in the WAN setting, and for 2 and 4 threads on a 1 Gbps LAN. However, SPD$\mathbb{Z}_{2^k}$ becomes computationally bounded in the case for one thread on the 1 Gbps LAN and for all number of threads we tested in the 10 Gbps LAN setting. This is visible from the graphs by noting the convergence of throughput of SPD$\mathbb{Z}_{2^k}$ in the WAN setting and at 2 threads in the 1 Gbps LAN.

For similar bit-lengths, the efficiency of SPD$\mathbb{Z}_{2^k}$ and MASCOT is almost the same. This is expected as our implementation is closely related to MASCOT. For smaller bit-lengths, *i.e.*, $k = 32$, our implementation is significantly more efficient since it requires far less communication. We note that the MASCOT implementation is hard-coded for fields of 128 bits and thus we cannot compare how it fares with a smaller field. Overdrive performs significantly better than SPD$\mathbb{Z}_{2^k}$ in the WAN setting, but the difference shrinks in a LAN. This is not surprising as Overdrive uses significantly less communication than MASCOT, and thus fares much better in a slower network than MASCOT, and consequently SPD$\mathbb{Z}_{2^k}$. SPD$\mathbb{Z}_{2^k}$ can nonetheless compete with Overdrive, given a fast enough network; (Fig. 1.4c) shows that the low bit setting for SPD$\mathbb{Z}_{2^k}$ matches Overdrive performance in a 10 Gbps LAN.

We ran SPD$\mathbb{Z}_{2^k}$ and MASCOT in batches of 1024 triples, and Overdrive in low-gear mode [96], the most efficient mode in the two-party setting. Increasing the thread count further did not significantly improve the throughput of any of the protocols we benchmarked.



(a) WAN (50 Mbps, 100 ms latency) (b) LAN (1 Gbps, 0.1 ms latency) (c) LAN (10 Gbps, 0.1 ms latency)

Figure 1.4: Triple generation throughput across different protocols and network settings.

The amount of preprocessed material needed for the operations/applications considered in this work can be found in Table 1.4. The table includes count of the arithmetic and bit triples needed for both SPD$\mathbb{Z}_{2^k}$ and SPDZ, along with the amount of random bits needed (which require an arithmetic multiplication triple for both SPD$\mathbb{Z}_{2^k}$ and SPDZ). We note the timing column is only an estimate, based on the time required for triple generation and bit triple generation. Thus the true time will be slightly larger for both SPD$\mathbb{Z}_{2^k}$ and SPDZ, because of the usage of authentication and input masks. However, these are in the order of a magnitude faster to construct compared to triples. Furthermore, the amount needed is fewer than the number of triples required and so the true impact of constructing these will be minuscule. Most importantly though, the amount required by both SPD$\mathbb{Z}_{2^k}$ and SPDZ is almost the same and so the effect on the relative difference between the two will be insignificant.

Table 1.4: Costs of the preprocessing for different operations/applications. Timings are estimates based on triples/random bits needed and are based on a 4 threads execution on a LAN supporting up to 10 Gbps. For SPDZ, Overdrive [96] is used. For bit triple generation the optimized TinyOT protocol by Wang *et al.* [129] is used.

	SPD$\mathbb{Z}_{2^k}$, $k = 32$, $\sigma = 26$				SPDZ, $k = 32$, $\sigma = 26$ (64 bit field)			
	# triples	# bit-triples	# random bits	time (ms)	# triples	# bit-triples	# random bits	time (ms)
Comparison	0	60	33	1.43	60	0	58	4.04
Equality	0	31	33	1.34	31	0	58	3.04
DTree (diabetes)	5460	15300	8415	571	20760	0	14790	1216
SVM (aloi)	63332	27720	15246	3055	91052	0	26796	4030

	SPD$\mathbb{Z}_{2^k}$, $k = 64$, $\sigma = 57$				SPDZ, $k = 64$, $\sigma = 57$ (128 bit field)			
	# triples	# bit-triples	# random bits	time (ms)	# triples	# bit-triples	# random bits	time (ms)
Comparison	0	124	65	7.22	124	0	121	14.9
Equality	0	63	65	7.04	63	0	121	11.2
DTree (diabetes)	5460	31620	16575	2417	37080	0	30855	4124
SVM (aloi)	63332	57288	30030	10006	120620	0	55902	10714

1.5.3 Applications

In tables 1.2 and 1.3 we show online benchmarking results of Protocols Π_{DecTree} and Π_{SVM} from Sec. 1.4. The tables show the online execution time of these protocols when obliviously classifying data, both using SPDZ and SPD$\mathbb{Z}_{2^k}$. For both decision tree and SVM evaluation, we measure evaluation time for a single data point, and the amortized time of evaluating multiple points in batches of 5 (since a service will likely classify more than a single data point).

Decision Trees

Table 1.2 shows online times for oblivious evaluation of some binary data models by De Cock *et al.* [45], based on datasets from the UCI repository[4]. The models are used to identify hills vs. valleys on 2-D graphs (Hill Valley), diabetes in women of Pima Indian decent (Diabetes) and spam vs. non-spam e-mail based on textual content (Spambase). We chose these models as they contain a large variation in the amount of features.

We see a noticeable, relative improvement of SPD$\mathbb{Z}_{2^k}$ over SPDZ in all the models we benchmarked, which further increases with the depth of the tree. As expected, batched evaluation yields better throughput; the batched runs also result in a bigger performance improvement for SPD$\mathbb{Z}_{2^k}$ over SPDZ. This shows that comparisons, which are needed for each node of the tree, become the bottleneck. This holds for both SPDZ and SPD$\mathbb{Z}_{2^k}$. Still, the impact is much greater for SPDZ as a depth increase from 3 to 9 results in a relative slowdown of up to 25x, whereas for SPD$\mathbb{Z}_{2^k}$ the slowdown is at most 18x. We thus see how important an efficient realization of an operation like comparison is for the real-world setting of decision trees. Finally, comparing $k = 32$ with $k = 64$ we see that the smaller ring gives up to a 1.9x improvement for SPD$\mathbb{Z}_{2^k}$ and 2.0x for SPDZ, showing the importance of flexibility in domain size.

[4]UC Irvine Machine Learning repository https://archive.ics.uci.edu/ml/datasets.html.

SVMs

Table 1.3 show oblivious evaluation of image classification models constructed by Makri *et al.* [104], and a model with few features but many classes[5]. The models by Makri *et al.* are built on the datasets CIFAR-10 [99] and MIT-67 [118] where Inception-v3 is used for feature extraction [126]. We chose these models to get a difference in number of classes and features. We see a large relative improvement of SPD$\mathbb{Z}_{2^k}$ over SPDZ. This holds even for a the smallest amount of classes, and thus smallest amount of comparisons as well. This indicates that the comparison is the main bottleneck in the SVM execution in both systems, as this factor is close to the direct improvement of comparison in SPD$\mathbb{Z}_{2^k}$ relative to SPDZ, as shown in Tables 1.1. It is interesting that this holds even for few classes and many features, as shown by the Cifar row in the batched setting.

1.6 Conclusions

In this work we showed how to compute basic functionality like comparison, equality, bit decomposition and truncation when working in the ring $\mathbb{Z}_{2^k}$, thus overcoming issues such as zero-divisors and lack of invertibility that arise in this setting.

We confirmed experimentally the conjecture from [47] that secure computation over the ring $\mathbb{Z}_{2^k}$ provides many advantages in the online phase, with only slight increase in offline cost. In particular we saw up to a 5-fold improvement in computation for various tasks, and up to a 85-fold reduction in online communication costs for secure comparison, as compared to the field setting.

In the future, we plan to explore other applications of SPD$\mathbb{Z}_{2^k}$, *e.g.*, neural network evaluation, where share conversions are known to help [105]. It also important to close the performance gap between SPD$\mathbb{Z}_{2^k}$ pre-processing and Overdrive; SHE-based techniques present a promising venue.

[5]The model aloi at `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#aloi`.

Chapter 2

Improved MPC Over Rings for Honest Majority

This chapter is based on the paper *Use your Brain! Arithmetic 3PC For Any Modulus with Active Security* [64], co-authored by SODA researchers, which is currently in submission.

2.1 Introduction

Secure Multiparty Computation (MPC) is an umbrella term for a broad range of cryptographic techniques and protocols that enable a set of parties $\mathrm{P}_1,\dots,\mathrm{P}_n$ to compute some function f of their private inputs $x_1,\dots,x_n$ without revealing anything beyond the output $f(x_1,\dots,x_n)$ of the computation. Most importantly, an actively misbehaving participant of the computation should not be able to bias the outcome of the computation (except by choosing their input) or learn anything about the inputs of the honest parties (except for what is leaked by the output itself). MPC started out as a purely theoretical research field 80s, but has recently developed into a science on the brink of practical deployment. This is witnessed by the constantly increasing number of real-world use cases, MPC framework implementations, and startups (see [12] for a survey).

The landscape of MPC protocols is broad and diverse, and protocols differ greatly depending on many parameters such as the number of involved parties, the corruption threshold, the adversarial model, and the network setting.

In this paper we focus on one of the most popular models for MPC, namely *three-party computation with an honest majority*. This model has been used in different real-world applications of MPC [30, 29, 25, 26, 7], often in the so-called *client-server* scenario where a possibly large number of clients secret share their inputs to three computation servers who can then perform the desired computation securely on their behalf [89] and return the result to the clients. A major advantage of the honest majority setting is that one can obtain protocols which do not rely on computationally expensive cryptographic operations such as exponentiations, oblivious transfer, etc, but typically only use light-weight arithmetic operations.

Existing implementations of three-party computation protocols for the honest-majority case fall into two broad categories: VIFF [52] and its successors [123] only support arithmetic computations over prime order fields. Sharemind's protocol suite [27, 28] can be used to evaluate arithmetic circuits with arbitrary word sizes, but is only secure against passive adversaries that follow the protocol faithfully. In practice this means that one has to either settle for rather weak security guarantees or one has to develop applications specifically tailored to rather unnatural word sizes instead of using the common 32- and 64-bit word sizes that dominate real-world system architectures. In particular, this means that a developer has to match the needs of the MPC framework it wants to use rather than the MPC framework matching the needs of the developer.

The main barrier to constructing actively secure protocols for evaluating arithmetic circuits with arbitrary word sizes lies in the fact that known approaches to achieving active security, like *information checking* techniques [119], require prime order fields. Up until recently it has been an open question to design protocols for arithmetic circuits with active security for arbitrary word sizes. In a recent work Damgård et al. [55] addressed this question by presenting a protocol compiler that transforms passively secure protocols into actively secure ones that can tolerate up to $\mathcal{O}(\sqrt{n})$ corruptions and only have a *constant* overhead in storage and computational work.

Our Contributions. In this paper we consider the class of protocols produced by compiler of Damgård et al. [55], and we improve such protocols in several ways. The main idea behind Damgård et al.'s compiler is to let the *real parties* "emulate" *virtual parties* that execute the desired computation on behalf of the real parties. The crucial point behind their compiler is that the virtual parties can execute[1] a passively secure protocol in a way that prevents any real party from actively misbehaving. Every time that a virtual party $\mathbb{P}_i$ is supposed to send a message to another virtual party $\mathbb{P}_j$ in the passively secure protocol, every real party that is emulating $\mathbb{P}_i$ computes the same message redundantly and sends it to every real party which is emulating $\mathbb{P}_j$. Each real party emulating $\mathbb{P}_j$ therefore receives a set of messages and aborts in case the received messages are not all equal. Intuitively this approach ensures active security as long as there is at least one honest party in the emulating set of every virtual party, since any malicious party either follows the protocol (in which case we effectively only have passive corruptions) or sends a message that disagrees with the message that is sent by at least one honest party (in which case the honest receiving party and consequently all other parties abort the protocol). This approach heavily relies on the fact that *all* protocol messages are sent redundantly, thus incurring a multiplicative blow-up in the bandwidth overhead of the protocol.

Our first contribution is of theoretical nature: we present an improved compiler that significantly reduces the number of redundant messages that need to be sent during a protocol execution. The idea behind our approach is to elect *one* real party in each virtual party to be the "brain", which sends all messages on behalf of its virtual party to *all* real parties in the receiving virtual party. The other real parties, the "pinkies", still receive messages from the brains and thus can locally follow the protocol execution. At the end of the protocol, right before the output of the computation is released, we then let all parties perform a single check that guarantees that all messages which were sent by the brains during the protocol are consistent with the messages all the pinkies would have sent. This check can be performed very efficiently by only checking consistency of the *hashes* of the protocol transcripts. It is clear that if any of the brains cheated during the protocol execution, then it must have sent a message that is inconsistent with the view of at least one pinky, thus the protocol would abort during the checking phase. On the downside our new compiler now imposes a stronger security requirement on the protocol it starts with. Note that honest brains continue the protocol execution up to the checking phase even if a malicious brain misbehaves, which means that we need a protocol that does not leak any private information even if cheating during the computation phase occurs. Thankfully, most passively secure secret sharing based protocols provide exactly the security guarantees that we need. More concretely, these protocols follow a compute-then-open structure, where the output of the computation is only revealed in the last round and any cheating during the

[1]Note that virtual parties do not physically exist. Whenever we say that "virtual parties execute a protocol", we really mean that the real parties simulate the virtual parties that execute the protocol.

preceding computation rounds can only affect the correctness of the output, but not the privacy of the inputs. Thus, by performing the consistency check described above at the end of the computation phase and *before* the output phase, we can ensure that no information is leaked. The security property sketched above has previously appeared in the literature under the name of weak privacy [73].

We formally present our new compiler and prove its security in Section 2.3. For the specific three-party case, our compiler produces a protocol, which is roughly twice as efficient as the protocol produced by the compiler of Damgård et al., since in the three party case each virtual party is emulated by one pinky and one brain.

Our second contribution is an improved preprocessing protocol for generated secret-shared multiplication triples. Damgård et al. generate both triples modulo a prime and triples modulo a power of 2, followed by a check-and-sacrifice step. We replace this by a preprocessing phase which does not perform any arithmetic in the larger prime field and solely uses computation modulo a slightly larger power of 2, thus improving on efficiency. While the sacrifice step is not performed in a field anymore, security follows using similar arguments as in the recent work on SPDZ over rings [47].

Finally we show that it is possible to completely avoid the preprocessing phase if one wishes to do so. Recall that our underlying protocols are assumed to preserve privacy until the outputs are opened. We exploit this security property by running the multiplication protocols optimistically and then, prior to opening the outputs, perform a single combined check. The two protocols, with preprocessing and with the postprocessing check, therefore offer different efficiency tradeoffs. The protocol with preprocessing has a leaner online phase, whereas the protocol with postprocessing has a better overall performances. Descriptions of our protocols are given in the full version of the paper [65].

In Section 2.4, we provide extensive performance benchmarks of our framework, both in the LAN as well as different WAN settings. Our protocols have been integrated in two of the leading MPC frameworks, namely the Sharemind MPC protocol suite and MP-SPDZ. As described in Section 2.4, we achieve the most efficient implementation of a three-party computation protocol for arithmetic circuits modulo 2^{64} with active security.

Other Related Work. The SPDZ family of protocols [21, 56, 54] efficiently implements MPC with active security in the *dishonest majority* setting. These protocols are split up into a slower, computationally secure *offline phase* in which correlated randomness in the form of so-called Beaver's triples is generated and a faster, information-theoretically secure *online phase* in which these triples are consumed to compute the desired functionality. Active security in the online phase is achieved using information theoretic message authentication codes (MACs), which until recently limited the SPDZ approach to computation over fields. In a recent work of Cramer et al. [47], this limitation has been lifted, allowing to perform computation modulo 2^k (by defining the MACs modulo to be $2^{k+\lambda}$ where λ is the security parameter, thus introducing an overhead proportional to the security parameter). An implementation (and optimizations) of [47] was presented in [50]. In addition [39] follows up [47] with a two-party protocol that uses homomorphic encryption and efficient zero-knowledge proofs in the precomputation phase.

Other recent works have considered active security in the three-party setting:

[69] uses correlated random number generation to achieve efficient preprocessing and replication to achieve active security. The protocol was originally presented only for Boolean circuits, but it was then later noticed that the approach generalizes to general rings [105]. They mention actively secure protocols in this setting, but do not give detailed protocol descriptions and only implement semi-honest versions of their protocols. For finite fields, [44] achieves active security by running two copies of the computation, respectively with real and random inputs, and uses the latter to verify

correctness (their approach can also be used for more than three parties).

A very different protocol for the same three party honest majority setting was presented in [42]. They combine two linear secret sharing schemes, one between two and other between three parties where the former is used to share a component of the latter sharing. This allows them to create a circuit dependent precomputation phase where all the two party sharings of random values are precomputed based on the circuit structure. The online phase focuses on computing modifiers to turn the random precomputed sharings into the desired values. Moreover, novel techniques for honest-majority MPC over rings have very recently been deployed in [3]. It is however still unclear whether this can lead to protocols which are efficient in practice.

2.2 Auxiliary Ideal Functionalities

We will make use of the following basic auxiliary ideal functionalities in this paper: The broadcast with *individual* abort functionality $\mathscr{F}_{\mathsf{bcast}}$ (Figure 2.1) allows a sender S to send a value v to a set of parties $\mathbb{P}$. The functionality guarantees that either a party aborts or it agrees on a consistent value with the other parties. Such a functionality is weaker than detectable broadcast [67], which requires that either all players agree on the same value or that all players unanimously abort. The functionality can easily be instantiated by letting the sender S send v to all parties in $\mathbb{P}$. Every party in $\mathbb{P}$ echoes the received value to all other parties in $\mathbb{P}$. Parties that receive consistent values output that value, parties that receive inconsistent values abort.

Functionality $\mathscr{F}_{\mathsf{bcast}}$

The functionality runs with sender S, who has input v, parties $P_1, \ldots, P_n$, and adversary $\mathscr{A}$.

1. S sends $(v, \mathbb{P})$ to $\mathscr{F}_{\mathsf{bcast}}$, where $v \in \{0,1\}^*$ and $\mathbb{P} \subset \{P_1 \ldots P_n\}$.
2. If either S or a party from $\mathbb{P}$ is corrupt, then $\mathscr{A}$ receives v and can decide which parties from $\mathbb{P}$ abort and which receive the output by sending a $|\mathbb{P}|$ long bit-vector b to the ideal functionality. For $P_i \in \mathbb{P}$:
   (a) If $b_i = 1$, then $\mathscr{F}_{\mathsf{bcast}}$ sends v to P_i.
   (b) If $b_i = 0$, then $\mathscr{F}_{\mathsf{bcast}}$ sends $\perp$ to P_i.

Figure 2.1: Broadcast functionality

The message checking functionality $\mathscr{F}_{\mathsf{check}}$ (Figure 2.2) allows a receiver, who holds a vector of messages, to check whether all other parties $P_1, \ldots, P_n$ hold the same vector of messages. The functionality can be instantiated by letting each party P_i sends its input to R. However, in this case the communication overhead would be $On\ \ell$ messages, where ℓ is the number of messages in a vector. Assuming the existence of collision-resistant hash functions, one can obtain a more communication efficient solution by simply letting all parties hash their message vectors into small digests before sending them to R. The communication overhead of such a solution would be $On\ \lambda$ bits if we assume that the output length of the hash function is $O\lambda$.

Functionality $\mathscr{F}_{\mathsf{check}}$

The functionality runs with receiver R, parties $\mathrm{P}_1, \ldots, \mathrm{P}_n$, and adversary $\mathscr{A}$. Party $\mathrm{P}_i \in \{\mathrm{P}_1, \ldots, \mathrm{P}_n\}$ has input $\left(m_{(1,i)}, \ldots, m_{(\ell,i)}\right)$ and receiver R has input $(m_1, \ldots, m_\ell)$.

1. All parties send their inputs to the ideal functionality.
2. $\mathscr{A}$ can decide whether to continue or to abort.
   (a) If $\mathscr{A}$ continues, then $\mathscr{F}_{\mathsf{check}}$ checks whether all message vectors are the identical. It outputs `same` if this is the case, and `different` otherwise, to the receiver R (in the latter case, the functionality outputs the inputs of all honest parties to $\mathscr{A}$).
   (b) If $\mathscr{A}$ aborts, then $\mathscr{F}_{\mathsf{check}}$ sends $\perp$ to all parties.

Figure 2.2: Message checking functionality

2.3 Extension of the Compiler by Damgård et al.

The compiler $\mathsf{COMP}_{\mathsf{old}}$ by Damgård et al. [55] takes an n-party passively $\left(t^2+t\right)$-secure protocol Π and transforms it into a protocol $\mathsf{COMP}_{\mathsf{old}}(\Pi)$ that is secure with abort against t active corruptions[2]. For example, for $t = 1$, the compiler can transform a passively two-secure three-party protocol into a protocol that is secure against one active corruption. The high-level idea of the compiler is to let virtual parties execute the passively secure protocol on behalf of the real parties. Each virtual party $\mathbb{P}_i$ is simulated by $t+1$ real parties $\mathrm{P}_i, \ldots, \mathrm{P}_{i+t}$ in a way that prevents an active adversary, who controls at most t real parties, from actively corrupting any of the virtual parties. In the following we will write $\mathrm{P}_j \in \mathbb{P}_i$ to denote that real party P_j is simulating virtual party $\mathbb{P}_i$.

The workflow of their compiler can be split into two phases. In the first phase, for each virtual party $\mathbb{P}_i$, all real parties $\mathrm{P}_j \in \mathbb{P}_i$ agree on a common input and randomness that will be used by $\mathbb{P}_i$ during the execution of the passively secure protocol Π. Having the same input and the same randomness, every $\mathrm{P}_j \in \mathbb{P}_i$ will be able to redundantly compute the exact same messages that $\mathbb{P}_i$ is supposed to send during the execution of Π. In the second phase, the virtual parties run Π to compute the desired functionality from the inputs and randomness that the virtual parties have agreed upon. Whenever $\mathbb{P}_i$ is supposed to send a message to $\mathbb{P}_j$ according to Π, *every* real party simulating $\mathbb{P}_i$ will send a separate message to *every* real party simulating $\mathbb{P}_j$. Each real party verifies that it receives the same message from all sending real parties and aborts if this is not the case.

Intuitively, the resulting protocol is secure against t active corruptions, since an adversary cannot misbehave on behalf of a virtual party it is simulating, and at the same time be consistent with at least one other honest real party that simulates the same virtual party.

From an efficiency point of view, every message from one $\mathbb{P}_i$ to some other $\mathbb{P}_j$ is sent redundantly from $t+1$ to $t+1$ real parties. That is, if the passively secure protocol Π sends ℓ messages during a protocol execution, then $\mathsf{COMP}_{\mathsf{old}}(\Pi)$ will send roughly $\mathscr{O}(\ell \cdot t^2)$ many messages.

[2]The authors also show how to achieve active security with guaranteed output delivery, but here we only focus on the case of security with abort.

2.3.1 A New Compiler for Protocols with Weak Privacy

We present a new compiler $\mathsf{COMP}_{\mathsf{new}}$, which makes slightly stronger assumptions about the starting protocol Π, but compiles it into an actively secure protocol in a more communication efficient manner. $\mathsf{COMP}_{\mathsf{new}}$ takes as input a (t^2+t)-weakly private protocol Π and outputs a compiled protocol $\mathsf{COMP}_{\mathsf{new}}(\Pi)$ that is secure against t active corruptions. If Π sends ℓ messages in total, then our compiled protocol will only send $\mathcal{O}(\ell \cdot t + t^2)$ messages.

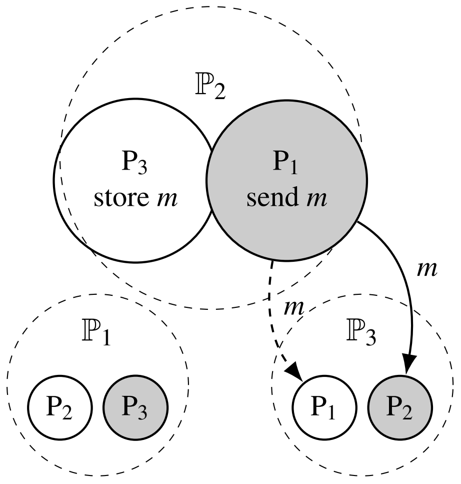


Figure 2.3: An illustration of our simulation strategy for the case of three parties with one active corruption. Dashed circles represent virtual parties. Solid circles inside the dashed circles represent the real parties that simulate the given virtual party. The brains of each virtual party are highlighted in gray. The figure illustrates the process of virtual party $\mathbb{P}_2$ sending a message to virtual party $\mathbb{P}_3$. The black arrows indicate that P_1, the brain of $\mathbb{P}_2$, sends one message to P_2 and one to P_1, which is omitted in reality, since it is sending a message to itself. P_3 stores this message in its transcript.

Our new compiler follows the approach of $\mathsf{COMP}_{\mathsf{old}}$. However, instead of verifying the validity of every single message between virtual parties as soon as it is sent, we will let the real parties simulate the virtual parties in a more optimistic and communication efficient fashion, where the correctness of all communicated messages is only verified once at the end of the computation phase, right before the opening phase of Π. Pushing the whole verification to the end of the computation phase allows us to reduce the total number or redundant messages that are sent. This new simulation strategy crucially relies on the weak active privacy of Π, since we are now allowing the adversary to misbehave up to the opening phase without aborting the protocol execution.

The first phase of $\mathsf{COMP}_{\mathsf{new}}$, where all parties agree on their inputs and random tapes, is identical to that of $\mathsf{COMP}_{\mathsf{old}}$ and is thus equally efficient. In the second phase, our new simulation approach works by selecting one arbitrary real party P_i in each virtual party $\mathbb{P}_j$ to be the brain $B_j := P_i$ of that virtual party. The brains will act on behalf of their corresponding virtual parties in an optimistic fashion and execute the computation phase of Π up to the opening phase. All other real parties, the pinkies, will receive the messages that their corresponding virtual parties should receive, which enables them to follow the protocol locally. However, the pinkies will not send any messages during the computation phase. They will only become actively involved in the opening phase to ensure that all brains behaved honestly during the computation phase. Once correctness is ensured, all parties will jointly perform the opening phase of Π. During the computation phase of Π, whenever virtual party

$\mathbb{P}_i$ is supposed to send a message to virtual party $\mathbb{P}_j$, we let B_i send one message to each real party in $\mathbb{P}_j$. The receiving real parties do not perform any checks at this moment and just store the message. B_j will optimistically continue the protocol execution on behalf of $\mathbb{P}_j$ according to Π and the received message. This simulation strategy is illustrated in Figure 2.3.

At the end of the computation phase, all real parties jointly make sure that for each pair $(\mathbb{P}_i, \mathbb{P}_j)$, the sending virtual party $\mathbb{P}_i$ always behaved honestly towards the receiving virtual party $\mathbb{P}_j$. This is accomplished by using a message checking protocol (that implements $\mathscr{F}_{\mathsf{check}}$). If any of these checks output different, then the protocol execution is aborted.

In the opening phase, after passing the previous check, every virtual party is supposed to send its last opening message to all other virtual parties. For each pair $(\mathbb{P}_i, \mathbb{P}_j)$, all real parties in $\mathbb{P}_i$ send the last message to all real parties in $\mathbb{P}_j$. Every receiving party checks that all $t+1$ received messages are consistent and aborts if this is not the case.

In our formal description, let $f(x_1, \ldots, x_n)$ be the n-party functionality that we want to compute. For the sake of simplicity and without loss of generality, we assume that all parties learn the output of the computation. Let $\mathbb{P}_i$ be the virtual party that is simulated by real parties $\mathrm{P}_i, \ldots, \mathrm{P}_{i+t}$. Let $\mathbb{V}_i$ be the set of virtual parties in whose simulation P_i participates. Let f' be a related n-party functionality that takes as input $\left(x_1^i, \ldots, x_n^i\right)$ from every P_i and outputs $f(\sum_{i=1}^n x_1^i, \ldots, \sum_{i=1}^n x_n^i)$. That is, every party inputs one secret share of every original input. The functionality f' reconstructs the original inputs for f from the secret shares and then evaluates f on those inputs. Let $\Pi_{f'}$ be a passively $\left(t^2+t\right)$-secure protocol with robust privacy that securely implements $\mathscr{F}_{f'}$. The formal description of our compiler is given in Figure 2.4. Throughout our description we assume that honest parties consider message that they do not receive as malicious and act accordingly.

Theorem 1. *Let $n \geq 3$. Assume $\Pi_{f'}$ implements n-party functionality $\mathscr{F}_{f'}$ with $\left(t^2+t\right)$-weak privacy. Then, $\mathsf{COMP}_{\mathsf{new}}\left(\Pi_{f'}\right)$ implements functionality $\mathscr{F}_f$ with active security under individual abort against t corruptions. If $\Pi_{f'}$ has a total bandwidth cost of ℓ messages, then $\mathsf{COMP}_{\mathsf{new}}\left(\Pi_{f'}\right)$ has a total bandwidth cost of $\mathscr{O}(\ell \cdot t + t^2)$ messages.*

Remark. Similar to Damgård et al. [55], we prove our result for the case of active security with individual abort, where some honest parties may terminate, while some may not. As in their work, our result easily extends to unanimous abort with one additional round of secure broadcast.

2.4 Implementation and Evaluation

To help adoption and accessibility of our protocols, we implemented them using Sharemind [24] and the MP-SPDZ framework [107]. We provide extensive benchmarks in both LAN and WAN settings for both implementations as well as a theoretical analysis of the asymptotic communication. Throughout this section, we use a statistical security parameter $\lambda = 40$.

2.4.1 Communication

Table 2.1 shows the communication complexity per multiplication in $\mathbb{Z}_{2^{64}}$ with the various protocols for statistical security parameter $\lambda = 40$. While the numbers are obtained from running the protocols in batches of at least one million with rounding, they match the asymptotic cost one would expect from a manual analysis. For comparison, we have added the figures reported in a recent concurrent work by Chaudhari et al. [42] (averaged over the parties because their protocol is asymmetric).

$\mathsf{COMP}_{\mathsf{new}}\left(\Pi_{f'}\right)$

Inputs: Each party P_i has input x_i.

---

1. **Input sharing:**
   (a) Each P_i secret shares its input $x_i = x_i^1 + \cdots + x_n^i$.
   (b) For $1 \le j \le n$, each P_i sends $\left(x_i^j, \mathbb{P}_j\right)$ to the broadcast functionality $\mathscr{F}_{\mathsf{bcast}}$.
   (c) Each P_i receives $z_j := \left(x_1^j, \ldots, x_n^j\right)$ for each $\mathbb{P}_j \in \mathbb{V}_i$ from the broadcast functionality and aborts if any of the shares equals $\perp$.
2. **Randomness:** Each brain B_i chooses a uniformly random string r_i and sends $(r_i, \mathbb{P}_i)$ to $\mathscr{F}_{\mathsf{bcast}}$. The receiving real parties abort if they receive $\perp$.
3. **Computation phase:** All virtual parties jointly execute the computation phase of $\Pi_{f'}$, where each $\mathbb{P}_i$ uses input z_j and random tape r_i, as follows:
   - Whenever $\mathbb{P}_i$ is supposed to send message m to $\mathbb{P}_j$, the brain B_i sends m to all real players in $\mathbb{P}_j$.
   - Whenever $\mathbb{P}_j$ receives message m, all pinkies store the message and only B_j continues the protocol according to $\Pi_{f'}$. The pinkies locally follow the protocol and compute the message that they would send.
4. **Check:** At the end of the computation phase, all parties, brains and pinkies, jointly check that the current transcript is correct. For each pair $(\mathbb{P}_i, \mathbb{P}_j)$, for each party $\mathrm{P}_k \in \mathbb{P}_j$, we invoke $\mathscr{F}_{\mathsf{check}}$, where P_k acts as the receiver and $\mathbb{P}_i$ act as the remaining parties. The input of P_k is the list of messages it received from $\mathbb{P}_i$ and the input of all parties from $\mathbb{P}_i$ is the list of messages that they would have sent. If any invocation outputs different, then the protocol execution is aborted.
5. **Opening phase:**
   (a) For each pair $(\mathbb{P}_i, \mathbb{P}_j)$, all real parties in $\mathbb{P}_i$ send the last message of $\Pi_{f'}$ to all real parties in $\mathbb{P}_j$.
   (b) Every real party in $\mathbb{P}_j$ checks that all received messages are equal. If they are it obtains the output of the computation and otherwise it aborts.

Figure 2.4: Formal description of our compiler.

One optimistic multiplication in $\mathbb{Z}_{2^m}$ requires sending m bits, and using Beaver multiplication in the data-dependent phase requires opening two masked values, thus sending $2m$ bits. A CDE+18-style sacrifice [47] requires two optimistic multplications and one opening in $\mathbb{Z}_{2^{m+\lambda}}$, while ABF+17 [10] asymptotically requires three optimistic multiplications and two classic sacrifices that require two

	Preprocessing	Data-dep.	Total
DOS18 preprocessing (single)	992	128	1120
DOS18 preprocessing (batch)	464	128	592
ABF+17 preprocessing	448	128	576
CDE+18 preprocessing	312	128	440
Postprocessing	-	312	312
Semi-honest	-	64	64
Malicious ASTRA [42]	448	85	553

Table 2.1: Communication per party and $\mathbb{Z}_{2^{64}}$ multiplication (bits)

openings each.[3] This comes down to $7m$ bits. Finally, DOS18 preprocessing [55] with SingleVerify requires two optimistic multiplications in $\mathbb{Z}_p$ and two openings in $\mathbb{Z}_p$ as well as sending two $m+\lambda$-bit values for sharing over the integers, totalling in $2(m+\lambda)+3\log p$ bits. It roughly holds that $\log p > 7+2m+3\lambda$, so for our choice of parameters $\log p > 255$.[4] The slight difference to the figure in the table comes from rounding up to multiples of eight.

2.4.2 Benchmarks

We have run our implementations in SIMD fashion, that is, combining the communication of a varying number of multiplications in as few network messages as possible. It is of little surprise that up to a certain number the throughput increases.

Figure 2.5 shows our benchmarks for various numbers of parallel multiplications in a LAN, that is AWS `c5.9xlarge` instances in the same region. This type features 36 virtual CPUs, 72 GiB of RAM, and 10 Gbit/s network network connection.

All benchmarks in this section are averages over ten executions. The figure for cut-and-choose is limited to 1048576 because the analysis by Araki et al. [10] mandates batches of at least this size. The plot shows that all malicious protocols perform similarly except the [55] protocol, and that the postprocessing protocol is slightly ahead as one would expect.

Figure 2.6 shows our benchmarks for various numbers of parallel multplications in a continental WAN, that is one AWS `c5.9xlarge` instance in each of Frankfurt, London, and Paris. The results mirror the results in the LAN setting except for the fact that 2^{20} parallel multiplications perform better than 2^{15} for all protocols. This is most likely because of the increased network delay of up to 12 ms.

Finally, 2.7 shows our benchmarks for a global WAN, that is one AWS `c5.9xlarge` instance in each of Frankfurt, Northern California, and Tokyo. The largest network latency we observed is 236 ms in this setting.

[3]Because of cut-and-choose we cannot use the trick used for DOS18-style sacrificing.

[4]According to Damgård et al. [55], $p > 100 \cdot 2^{2m+2\lambda}$, but a quick recalculation of $24 \cdot B^2 2^\lambda$ with $B = 2^{m+\lambda+1}$ shows that it should be $3s$ instead of 2λ in the inequality for p.

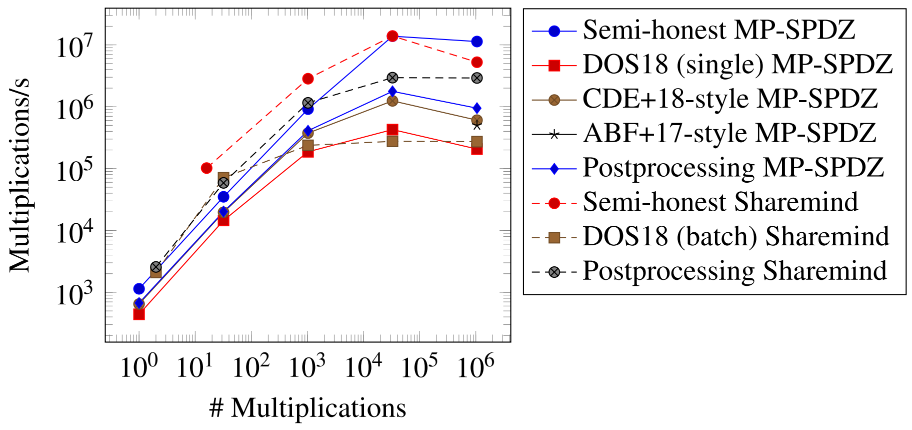


Figure 2.5: Throughput in LAN

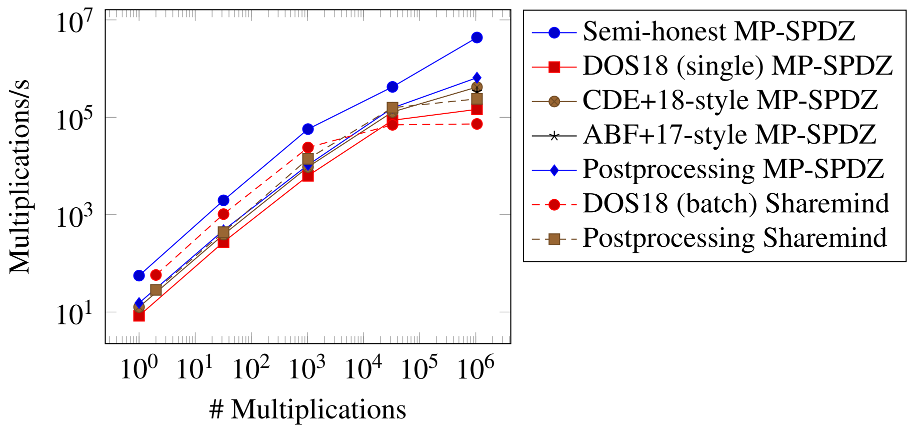


Figure 2.6: Throughput in continental WAN

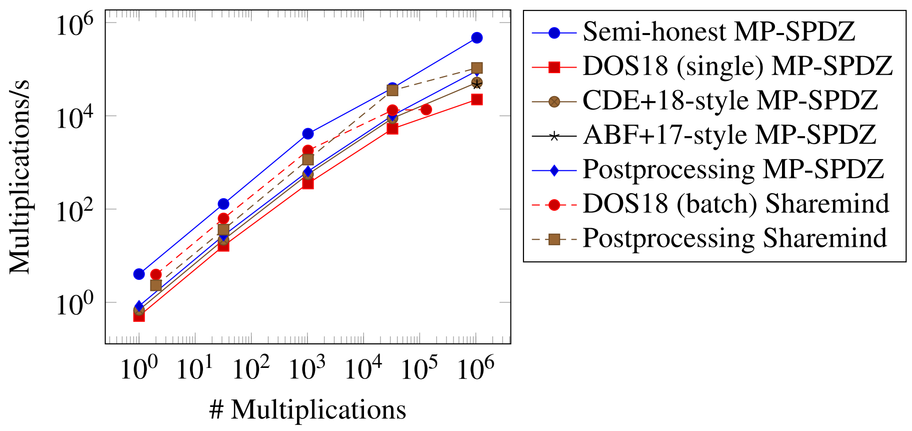


Figure 2.7: Throughput in global WAN

Chapter 3

Secure Evaluation of Quantized Neural Networks

This chapter is based on the paper *Secure Evaluation of Quantized Neural Networks* [15], co-authored by SODA researchers, presented at two workshops on Privacy-Preserving Machine Learning, co-located with Crypto 2019 and CCS 2019.

3.1 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 possible solution for many real-life problems such as facial recognition [100], image and video analysis for self-driving cars [31] and even for playing boardgames (most readers probably know of *AlphaGo* [125] which in 2016 beat one of the best Go players currently alive, Lee Sedol [131]). CNNs have also found applications within areas of medicine. [66], For example, demonstrates that CNNs are as effective as experts at detecting skin cancers from images.

In many applications the data on which the prediction is performed is sensitive and ideally no information should be disclosed to the model owner. Conversely, the model owner may have invested a significant amount of time and effort into developing his model and so might not like to disclose his model to the input owner. MPC arises as a natural solution to this dilemma. Indeed, the security guarantees that MPC provide state exactly that participants learn nothing except their respective inputs and the output.

While secure inference has been an active research topic within the last couple of years (see e.g., [106, 105, 91, 103]), previous solutions often fall short of the goal when it comes to developing solutions that are both efficient and practically usable. Evaluating a Neural Network is quite expensive in terms of the operations. For example, evaluating a single convolutional layer with a $d_i \times h_f \times w_f \times d_f$ filter and $h_i \times w_i \times d_i$ input takes in the order of $h_f h_i w_f w_i d_f d_i$ operations; and if both the filter and input are represented in floating point, this quickly becomes prohibitive to do securely (e.g., in MPC) for even small inputs and filters. Moreover, each linear layer (such as a convolution) is followed by a non-linear one, such as $\mathsf{ReLU}(x) = \max(0,x)$, which is expensive too when computed securely.

For these reasons, previous works resort to modifying the NN models in some way in order to obtain a representation that is “MPC friendly” (or GC/FHE friendly). However, such changes carry with them several serious flaws that hurt the validity of these works and in particular their practical applicability (especially among non-experts). The central theme of our work is addressing these problems and so we discuss them in more detail here. We focus our discussion on secure inference using MPC, however all our arguments apply to Garbled Circuits or Fully-Homomorphic Encryption based solutions as well.

Floating vs. fixed point numbers. MPC protocols support simple arithmetic (or boolean) operations such as addition and multiplication over the integers (or XOR and AND over $\mathbb{Z}_2$). However, they lack efficient support for operating on floating point numbers and so performing floating point arithmetic is typically done by operating on the level of bits (i.e., by evaluating a circuit that performs the operation). Using fixed point numbers instead can speed up the computation quite a bit and so representing a model's parameters as fixed point numbers is typically what is done in the Secure Inference literature.

The question is: what effect does this switch in representation have on the model? Most networks perform well on very simple problems such as MNIST and CIFAR10, which are the benchmarks that is usually run, and it is not unreasonable to expect the models to still perform well even after a change in representation. Moreover, the experimental results we see in previous work (e.g., [106]) that evaluates such models show that the effect of going from floating to fixed point numbers is essentially negligible.

However, these results only show that the modifications work for *those specific models*; in particular, it is not at all clear that these modifications have similar benign effects on networks that are much larger and much more complex. In fact, the research area of *quantization* (which we will return to in the next section) deal with this exact question.

Changing activation functions. Closely related to the discussion in the previous paragraphs is the question about the effect of changing the activation functions of a model. Naturally, the same issues exist here: it is not clear if a result in one model generalizes to another. This argument applies to works that approximate activation functions [106, 91], as well works that replace activation functions by "similar" functions (e.g., using max pooling instead of average pooling, or square instead of ReLU).

Existing frameworks. Finally, altering models as described above is very often particular to a specific secure protocol and so it is rarely clear how to train and evaluate a model that will perform well when run securely. This creates an unnecessary entanglement between obtaining the input for the secure protocol and the secure protocol itself, which is an obstacle for practical deployment. For example, this entanglement creates an unreasonable barrier to non-experts (on the secure protocol) who wish to run the protocol in practice. It should ideally be the case that one can train a model using e.g., Tensorflow [1] or PyTorch [113] (or some other popular training framework) and then use the model so obtained directly as input, without having to worry about a loss in accuracy or some other incompatibility.

3.1.1 Related work

We provide here a brief overview of prior work on secure inference with an emphasis on modifications that are applied to the models they evaluate.

Early work on evaluating NNs securely can be traced back at least to the work by Barni et al. [17] and Orlandi et al. [112]. Both works use Homomorphic Encryption to evaluate the networks, and the implementation in [112] use a fixed-point approximation. In fact, all prior work that uses some variant of Homomorphic Encryption that we are aware of rely on a conversion to fixed-point. CryptoNets by Gilad-Bachrach et al. [76] chooses to approximate the max pooling function, and uses a square activation function (i.e., $f(x) = x^2$) that was observed to behave badly during training on which the authors note that it may not be suitable for training larger networks. The authors of [41] recognize this issue and instead approximate the ReLU activation using polynomials. This too is problematic unless batch normalization [84] is used.

Several works exist that make use of the protocol switching idea of ABY [58]. MiniONN [103] provides a framework for transforming a previously trained model into one that can be run with ABY. While this is step in the right direction, in terms of usability, it still falls victim to the issue of having to modify the model (and thus requiring users to be aware of the transformation when the model is trained). Later, the authors of [121] present another solution that also uses ABY, but which introduces a third party in order to compute Oblivious Transfers more efficiently. Their protocol enjoys a 3.5x speedup over [103]. Lastly, ABY3 [105] (essentially ABY for 3 parties) also provide micro benchmarks for operations common in NNs.

SecureML [106] and SecureNN [128] both use fixed point arithmetic in MPC and consider training as well as inference. SecureML introduces an optimized way of performing the truncation necessary when performing a fixed-point multiplication; however, it only works for a small amount of multiplications (and the number of multiplications one can perform using this technique is not clear).

DeepSecure [122] take a Garbled-Circuit-only approach and implement a library for various common operations that are relevant for NN evaluation. Recently, and concurrent to our work, Riazi et al. [120] presented XONN, which is an efficient way of evaluating NNs using garbled circuits. Their results can be attributed to the fact that they use *binarized* models [83]—a form of quantized models—which all weights in a model into values $-1/1$.

In another related work by SODA resesarchers, Abspoel et al. [2] designed a novel small-range comparison protocol and applied this to efficient, secure evaluation of binarized NNs, similarly to XONN.

3.1.2 Quantization in prior work

With the exception of DeepSecure [122] and XONN [120], all works listed in the previous section employ a naive floating point to fixed point transformation.

The XONN work deserves a special mention here as they rely on a technique *from the Machine Learning literature*. In particular, they make the same observation as us that some of the challenges that is faced when constructing protocols for secure inference have in fact already been considered in the ML literature.

3.2 Quantization

The goal of model quantization is firstly to reduce the size of the model, and secondly to reduce the computational load of evaluating the model (e.g., by requiring only integer operations and binary shifts). Reducing the model size and complexity is an attractive goal if one wants to use a model on a device with little computational power, such as smartphones, IOT or embedded devices.

We remark that quantization is only applied to the forward pass—or inference step—and not during training, and so our approach only applies here. However, ML researches are slowly looking at performing model training with a quantization flavor to it, so investigating secure training of quantized models is a promising direction for future work.

While the quantization scheme we focus on in this work uses many similar techniques to those we discussed in the introduction (namely, fixed point arithmetic), we stress that the techniques we use here were not developed with the security protocol in mind, and moreover, these were developed by a team of ML researchers with the explicit aim of preserving the accuracy of the models. The ultimate value of this is that, even if their methods can still be considered heuristics, these are backed up by an extensive body of research made by competent researchers in the area. Moreover, given that these tools find applications beyond secure inference (namely, what they were designed for initially), their validity is likely to be assessed more thoroughly by a broader community.

3.2.1 TFLite Quantization

For $\boldsymbol{x} \in \mathbb{R}^{h_i \times w_i \times d_i}$ we use $\boldsymbol{x}[i,j,k] \in \mathbb{R}$ to denote taking the i'th value across the first dimension, the j'th value across the second, and the k'th value across the last. Similarly, $\boldsymbol{x}[\cdot,\cdot,k] \in \mathbb{R}^{h_i \times w_i}$ denotes the matrix that is obtained by fixing a particular value for the third dimension of $\boldsymbol{x}$.

For a filter $\boldsymbol{w} \in \mathbb{R}^{d_i \times h_f \times w_f \times d_o}$, bias $\boldsymbol{b} \in \mathbb{R}^{d_o}$ and input $\boldsymbol{x} \in \mathbb{R}^{h_i \times w_i \times d_i}$ it suffices to observe that output entry $\boldsymbol{y}[i,j,k]$ can be computed as $\boldsymbol{y}[i,j,k] = \mathsf{dot}(\boldsymbol{x}_{i,j}, \boldsymbol{w}[k,\cdot,\cdot,\cdot]) + \boldsymbol{b}[k]$, where $\boldsymbol{x}_{i,j} \in \mathbb{R}^{h_f w_f d_i}$ vector derived from $\boldsymbol{x}$.[1]

We consider the activation function $\mathsf{ReLU6}(x) = \min(6, \max(0,x))$, which is used as the activation function in the MobileNets family of networks. However, the techniques presented in this section also work for the more general ReLU.

Getting rid of reals

The scheme of Jacobs et al. [88] can be seen as a type of shifted fixed point encoding. Let α, s be values of $\mathbb{R}$ and z, a values of $\mathbb{Z}_{2^8}$ (i.e., bytes) and define the function $\mathsf{dequant}_{s,z}(a) = s(a-z) = \alpha'$. We call s the *scale* and z the *zero-point*. This procedure effectively maps the interval $[0, 2^8-1]$ injectively into the discrete $I = [-s \cdot z, s(2^8-1-z)]$. We define the quantization of a value $\alpha \in I$ as $\mathsf{quant}_{s,z}(\alpha')$ where α' is the closest number to α that is in the image of $\mathsf{dequant}_{s,z}$.

Efficient Inference

We now describe how the quantization scheme just described allows us to compute dot-products (and thus convolutions) using only integer operations and a single truncation. Furthermore, we describe how the $\mathsf{ReLU6}$ activation is subsumed by the truncation in the dot-product computation, effectively meaning can compute $\mathsf{ReLU6}$ free of cost. (The trick applies to any ReLU-like activation function provided it is clipped to an interval.)

Convolutions. Let $\boldsymbol{\alpha} = (\alpha_1, \dots, \alpha_N)$, $\boldsymbol{\beta} = (\beta_1, \dots, \beta_N)$ be two real-valued vectors, and let s_α, z_α respectively, s_β, z_β be their quantization parameters. Let $\gamma = \sum_i^N \alpha_i \beta_i$ be the real-valued output and s_γ, z_γ its quantization parameters. Finally, let $a_i = \mathsf{quant}_{s_\alpha, z_\alpha}(\alpha_i)$, $b_i = \mathsf{quant}_{s_\beta, z_\beta}(\beta_i)$ and $y = \mathsf{quant}_{s_\gamma, z_\gamma}(\gamma)$. Observe that

$$s_\gamma(y - z_\gamma) \approx \gamma = \sum_{i=1}^{N} \alpha_i \beta_i \approx \sum_{i=1}^{N} s_\alpha(a_i - z_\alpha) s_\beta(b_i - z_\beta)$$

[1] $\boldsymbol{x}_{i,j}$ is the sub-tensor centered around $\boldsymbol{x}[i,j,\cdot]$, that has been flattened rows first then depth.

And so, by rewriting, we can compute y by

$$y = z_\gamma + \frac{s_\alpha s_\beta}{s_\gamma} \sum_{i=1}^{N} (a_i - z_\alpha)(b_i - z_\beta). \tag{3.1}$$

Notice that the computation above requires only integer operations and a single multiplication by the real value $s = (s_\alpha s_\beta)/s_\gamma$. Tensorflow handles this multiplication by representing s as a fixed-point value, which allows computing the final value as an integer multiplication followed by a shift. More precisely, write $s = 2^{-n}s'$ where $s' \in [0.5, 1)$. Next, s' is encoded as $s'' \approx 2^{-31}s'$ where s' is a 32-bit integer. Notice that this approximates s' with at least 30 bits of accuracy.

1. Compute $t = \sum_i^N (a_i - z_\alpha)(b_i - z_\beta)$,
2. Encode $s = (s_\alpha s_\beta)/s_\gamma$ as described above. Compute $t' = t \cdot s'$ and then $t' \cdot 2^{-n-31}$.
3. Finally, t' is rounded to the nearest integer and z_γ is added.

As a final thing, we note that the quantization parameters of the bias term needs to satisfy that $z_{\mathsf{bias}} = 0$ and $s_{\mathsf{bias}} = (s_\alpha s_\beta)/s_\gamma$, since otherwise we would need to perform another floating point multiplication.

Efficient activation functions. By being clever about the values of s and z, it is possible to skip computing $\mathsf{ReLU6}$ altogether, a feature that provides a considerable speedup over having to compute the max function (i.e., comparison). Recall that $\mathsf{ReLU6}(x) = \max(0, \min(6, x))$, that is, the domain of $\mathsf{ReLU6}$ is the interval $[0, 6]$. However, this implies that by setting $z = 0$ and $s = 1/255$, we have that the domain of $\mathsf{dequant}_{s,z}$ is in the domain of $\mathsf{ReLU6}$. In particular, $\mathsf{ReLU6}(\mathsf{dequant}_{s,z}(x)) = \mathsf{dequant}_{s,z}(x)$ for all $x \in \mathbb{Z}_{2^8}$.

3.3 Protocol

Our system model is the standard outsourced computation model with three servers. That is, we consider a setting with three parties P_1, P_2 and P_3 among which at most one is corrupted. During the input phase, the model owner secret shares its model among each of the servers and the input owner does the same with their input. In the end, each server sends their shares of the result to the input owner who in this way obtains the result of evaluating the model on their input. We use the passively secure protocol by Araki et al. [11], which is based on replicated secret sharing, and we primarily employ it in the ring defined by arithmetic modulo 2^{64}. We choose to use a ring like $\mathbb{Z}_{2^{64}}$ instead of a field, as usual in MPC, motivated by the fact that such rings have performance benefits in MPC. In this protocol a value $x \in \mathbb{Z}_{2^{64}}$ is secret-shared among P_1, P_2, P_3 if each P_i holds a random pair (x_i, x_{i-1}) (indexes wrap modulo 3) such that $x_1 + x_2 + x_3 = x \bmod 2^{64}$. We denote this sharing by $[x]$. Clearly, if the adversary only controls one party passively, it cannot learn anything about x.

3.3.1 Protocol details

Secure Addition and Multiplication. The secret-sharing scheme used is additively homomorphic, which means that the parties can obtain shares of the sum of two shared values by simply adding locally their shares. Furthermore, multiplications can be computed with passive security by letting each party send only one ring element, as described in [11] (and simplified in [101]).

Dot-Products. A central operation in any CNN is to run dot products(e.g. for the matrix multiplications). When each one of the entries of the input vectors is secret-shared, this would involve n times the communication of a single secure multiplication, where n is the length of the vectors. Fortunately, as observed for example in [44], the multiplication protocol from [11] allows such dot-products to be computed at the communication cost of a single multiplication.

Secure Truncation.

We provide a novel method to truncate a shared value by an *unknown* amount of bits, which is shared as well. This is contrast to the typical scenario addressed in MPC which involves truncating a shared value by a known amount (especially useful for fixed point arithmetic). The idea is to turn the truncation by a secret amount (aka right-shift), into a multiplication by a secret power of 2 (aka left-shift) followed by a regular truncation. Put differently, in order to obtain a share of $\lfloor 2^{-k} \cdot x \rceil$ where both x and k are secret shared, we instead compute $2^{K-k} \cdot 2^{-K} \cdot x$ where $K \geq k$ is an upper bound on k and is public. While this method does require us to use a modulus that is slightly bigger than 64 bits, the effect on the efficiency is minor. In the following $[\cdot]$ denotes a sharing as used in the MPC. This protocol proceeds in a number of steps. First we need to compute $[2^{K-k}]$ given $[K-k]$; next is to compute $[\lfloor 2^{-K} \cdot x \rceil]$ where K is public. We also need a way to reduce a secret shared value by a power of two, i.e., given $[x]$ we need to be able to compute $[x \mod 2^k]$.

Reducing a share modulo a power of 2. Suppose parties have a sharing $[x]$ and wish to compute $[y] = [x \mod 2^k]$ for a public k. First step is to compute $[x'] = [x \ll (\ell - k)]$ where 2^ℓ is the modulus that is used in the MPC (e.g., 64). Note that this is a local operation as it corresponds to multiplying each share by $2^{\ell-k}$. Next step is to mask $[x']$ using $k' = \ell - k$ random bits. Reveal the result $c = [x'] + 2^k \sum_{i=0}^{\ell-k} [b_i] 2^i$. Shift c to the right by k positions and subtract $\sum_{i=0}^{\ell-k} [b_i] 2^i$ to obtain the result $[y]$. Finally, we need to account for a potential wrap around and so need to compute a comparison between $c \gg k$ (a public value) and $\sum_{i=0}^{\ell-k} [b_i] 2^i$ (a secret value). This comparison can be computed as a binary circuit and we subtract $2^{\ell-k}$ from $[y]$ if necessary.

Computing $[2^{K-k}]$. Given $[K-k] = K - [k]$ we wish to compute $[2^{K-k}]$. To do so, we first bit-decompose $[K-k]$ into bits $[b_i]$ such that $[K-k] = \sum_i [b_i] 2^i$. The desired sharing can then be computed as the product $[2^{K-k}] = \prod_i (1 + [b_i](2^{2^i} - 1))$.

Truncation by a public value. Note that $\lfloor 2^{-K} \cdot x \rceil = \lfloor 2^{-K} \cdot (x + 2^{K-1}) \rfloor$[2] Thus it is enough to compute a regular right-shift. We first compute $[x'] = [x \mod 2^K]$ using the technique described above. Define $[y] = [x - x']$ and note that this corresponds to zeroing the bottom K bits of x; i.e., the bits that we wish to "shift out", so to speak. The rest is now very much like the protocol for reducing modulo a power of 2: Mask the top $\ell - K$ bits, open the result, shift to the right and undo the masking taking the potential overflow into consideration.

3.4 Experimental Results

In the full version of this work we run a series of benchmarks that include micro-operations, evaluation of large MobileNet models, and comparison to previous work.

[2]This is simply using the fact that $\lfloor x \rceil = \lfloor x + 1/2 \rfloor$.

We include in this abstract the evaluation of full-sized networks in the MobileNet family. Some information about these models can be seen in Table 3.1. We present benchmarks for secure inference modulo 2^{64} (3.2) and modulo a prime of roughly 64 bits (3.3), and also for active security modulo this prime. The goal of this is to provide an overview of the different factors that affect the performance of the computation, like the type of security, the algebraic structure, and the desired accuracy.

We remark that the decision to benchmark our framework on *existing* models is only for the sake of simplicity. Given the tools provided by Tensorflow, it would have been also possible to train a floating-point model in Tensorflow and then convert it to a quantized version of it by using the TOCO converter from TFLite. We believe that this feature is an important step towards the deployment of secure inference in realistic scenarios.

Name	Top-1 (%)	Top-5 (%)	Name	Top-1 (%)	Top-5 (%)
0.25_128	39.5	64.4	0.75_128	55.9	79.1
0.25_160	42.8	68.1	0.75_160	62.4	83.7
0.25_192	45.7	70.8	0.75_192	66.1	86.2
0.25_224	48.2	72.8	0.75_224	66.9	86.9
0.50_128	54.9	78.1	1.00_128	63.3	84.1
0.50_160	57.2	80.5	1.00_160	66.9	86.7
0.50_192	59.9	82.1	1.00_192	69.1	88.1
0.50_224	61.2	83.2	1.00_224	70.0	89.0

Table 3.1: Top 1 and top 5 accuracy of the different models. Each model name is of the format d_s where d is a multiplier that essentially scales the number of parameters in each layer (thus, smaller d implies a model that is easier to evaluate). s on the other denotes the size of the input, e.g., $s = 128$ means that inputs are 128×128 images. See https://github.com/tensorflow/models/blob/5fd32ef62e37a8124bf8849f7bea65fbd8cd7bdd/research/slim/nets/mobilenet_v1.md

Name	Runetime (s)	Comm. (gb)	Name	Runetime (s)	Comm. (gb)
0.25_128	1.7	1.7	0.75_128	5.0	5.0
0.25_160	2.6	2.6	0.75_160	7.8	7.8
0.25_192	3.6	3.7	0.75_192	10.9	11.2
0.25_224	5.0	5.1	0.75_224	15.3	15.3
0.50_128	3.4	3.3	1.00_128	7.4	6.7
0.50_160	5.1	5.2	1.00_160	10.6	10.4
0.50_192	7.2	7.5	1.00_192	20.6	20.4
0.50_224	10.0	10.2	1.00_224	25.9	27.0

Table 3.2: Runtime and communication for securely evaluating various models with a protocol that uses mod 2^{64}.

3.5 Conclusion

Our work shows that securely evaluating a large and meaningful network using MPC, without modifying its accuracy substantially, is practically possible. We also show that some techniques that already

	passive		active	
Name	Runetime (s)	Comm. (gb)	Runetime (s)	Comm. (gb)
0.25_128	7.1	4.4	22.3	12.9
0.25_160	10.9	6.8	34.3	20.1
0.25_192	15.7	9.8	49.0	28.9
0.25_224	21.2	13.3	66.3	39.3
0.50_128	14.0	8.7	47.5	28.2
0.50_160	21.7	13.6	73.8	44.0
0.50_192	30.9	19.5	105.5	63.3
0.50_224	42.2	26.6	143.5	86.1
0.75_128	21.0	13.0	76.7	46.0
0.75_160	32.3	20.4	119.0	71.7
0.75_192	46.2	29.3	171.0	103.2
0.75_224	63.6	39.9	232.9	140.5
1.00_128	28.3	17.4	109.8	66.2
1.00_160	43.3	27.1	170.5	103.3
1.00_192	61.9	39.1	244.3	148.7
1.00_224	83.9	53.2	332.5	202.4

Table 3.3: Runetime and communication for model evaluation using a mod prime protocol for both active and passive security.

exist in the Machine Learning literature, namely quantization, can be useful for Cryptography researchers as well. Using these techniques we manage to evaluate securely very large networks whilst preserving their accuracy, with an efficiency that is already practical in many realistic scenarios. Furthermore, for the first time in the literature, we provide running times and memory usages for active security, which, in spite of being much less practical than the passively secure setting, illustrates that such a strong notion of security is within reach for some applications.

In general, we believe it is important for the cryptographic community to work closely to the machine learning community, and we see our work as an early step in this direction.

Chapter 4

Efficient Secure Ridge Regression from Randomized Gaussian Elimination

This chapter is based on the paper *Efficient Secure Ridge Regression from Randomized Gaussian Elimination* [22], co-authored by SODA researchers, presented at the TPMPC workshop and the workshop on Privacy-Preserving Machine Learning co-located with CCS 2019.

4.1 Introduction

Recent years have seen significant advances in privacy-preserving data mining and machine learning. Secure multiparty computation (MPC) is a promising type of cryptographic protocol for enhancing the security and privacy properties of existing data mining and machine learning algorithms. Handling large datasets, however, still poses practical challenges due to the overhead incurred by MPC.

Secure regression is a problem that received much attention as the resulting cryptographic protocols have the potential of handling relatively large datasets, see, e.g., [62, 79, 110, 75, 71]. When applied to linear and ridge regression, the overhead for MPC is limited because of the highly linear nature of the computation. The bulk of the computation consists of taking inner products, which can be done securely at low cost in many MPC frameworks.

In this paper we develop particularly efficient m-party protocols for ridge regression tolerating a dishonest minority of up to t passively corrupt parties, $0 \leq t \leq (m-1)/2$. We present a range of practical optimizations, which are combined into a very competitive solution for secure ridge regression. We present experimental results to support our claims using the MPyC framework for secure multiparty computation.

4.2 Approach

Ridge regression (or, Tikhonov regularization) is a classic problem in statistics. Nowadays, the problem is broadly studied and applied in machine learning, and many algorithms have been proposed covering various types and dimensions of input data. The popular tool Scikit-learn, for instance, provides six different solvers for ridge regression, most of which also use different approaches for sparse and dense data [114].

The solver used in the present paper directly uses the closed-form solution for ridge regression, cf. Eqs. (4.2) and (4.3). An alternative approach is to approximate the solution using an iterative solver, viewing ridge regression as an optimization problem minimizing (4.1). Well-known iterative solvers are stochastic gradient descent and its many variations (e.g., mini-batch gradient descent).

However, there are two major impediments for adopting iterative solvers in an MPC setting. Firstly, all arithmetic involves real-valued numbers, which needs to be approximated using secure fixed-point arithmetic (as secure floating-point arithmetic is simply too expensive). The use of secure fixed-point numbers incurs a substantial overhead and could lead to numerical stability issues. Secondly, one needs to control the number of iterations. In an MPC setting, evaluation of a stopping criterion may form a bottleneck in itself, and fixing the number of iterations beforehand may demand a high number of iterations (to ensure convergence for all inputs). The advantage of the iterative approach is that it generalizes immediately to related machine learning algorithms such as logistic regression and support vector machines. Adapting the computation of the gradient suffices to solve these problems as well.

As we show in this paper, there are major advantages to solving the ridge regression problem directly. It allows us to avoid fixed-point arithmetic entirely. Issues surrounding rounding errors are limited to the input phase, when real-valued inputs are converted to integral values using appropriate scaling. From that point on all computations are exact, using integer arithmetic only. The main issue left is the growth of the numbers, but we will show that even for huge datasets, our approach is practical and leads to very competitive results in an MPC setting.

The closed-form solution is in fact a matrix equation, which can in turn be solved directly or iteratively, as we will discuss in Section 4.6.

4.2.1 Roadmap

We present mathematical preliminaries in Section 4.3, and the basics on linear regression and ridge regression in Section 4.4. Next we introduce basic notation for MPC based on Shamir secret sharing in Section 4.5. In Section 4.6 we discuss the relevant choices for solving linear systems of equations in an MPC setting, showing how we avoid the use of rational reconstruction. Section 4.7 contains the basic protocols for secure linear algebra, which we use in our protocol for secure ridge regression in Section 4.8. Finally, we discuss the performance in Section 4.9 and conclude in Section 4.10.

4.3 Preliminaries

We use common notation for matrices and vectors. For $d \geq 1$, the group of $d \times d$ invertible matrices over a field $\mathbb{F}$ is denoted by $\mathrm{GL}_d(\mathbb{F})$. The groups of $d \times d$ lower resp. upper triangular invertible matrices are denoted by $\mathrm{L}_d(\mathbb{F}), \mathrm{U}_d(\mathbb{F}) \subseteq \mathrm{GL}_d(\mathbb{F})$, and we use $\mathrm{L}_d^1(\mathbb{F})$ to denote the group of lower triangular matrices with an all-ones diagonal.

A matrix $A \in \mathrm{GL}_d(\mathbb{F})$ is said to have an **LU-decomposition** if $A = LU$ for some $L \in \mathrm{L}_d^1(\mathbb{F})$ and $U \in \mathrm{U}_d(\mathbb{F})$. We use $\mathrm{LU}_d(\mathbb{F})$ to denote the set of all matrices in $\mathrm{GL}_d(\mathbb{F})$ that have an LU-decomposition. Note that the LU-decomposition for each $A \in \mathrm{LU}_d(\mathbb{F})$ is unique. Similarly, a matrix $A \in \mathrm{GL}_d(\mathbb{F})$ is said to have a **Cholesky decomposition** if $A = LL^{\mathsf{T}}$ for some $L \in \mathrm{L}_d(\mathbb{F})$. The Cholesky decomposition is also unique, and exists over $\mathbb{F} = \mathbb{R}$ if and only if A is symmetric and positive definite.

For $A \in \mathrm{GL}_d(\mathbb{F})$, we use $\operatorname{adj} A = \det(A)A^{-1}$ to denote the **adjugate** of A. For our approach, a key property is that if A is integral then so are $\det A$ and $\operatorname{adj} A$. That is, if A is a matrix over $\mathbb{Z}$, then $\det A \in \mathbb{Z}$ and $\operatorname{adj} A$ is also a matrix over $\mathbb{Z}$. Furthermore, **Hadamard's inequality** states that $|\det A| \leq \prod_{i=1}^{d} \|\boldsymbol{a}_i\|_2$, where $\boldsymbol{a}_i$ are the rows (or columns) of A. For $\alpha = \|A\|_{\max}$, Hadamard's inequality implies $|\det A| \le d^{d/2}\alpha^d$. If A is symmetric and positive definite, $\det A$ is positive and Hadamard's inequality becomes $\det A \le \prod_{i=1}^{d} a_{i,i}$ and we get $\det A \le \alpha^d$. Finally, Hadamard's inequality yields $\|\operatorname{adj} A\|_{\max} \le (d-1)^{(d-1)/2}\alpha^{d-1}$ as bound for the adjugate.

Gaussian elimination and variations thereof are used to compute $\det A$, $\operatorname{adj} A$, and A^{-1}. For example, A^{-1} is computed by transforming the augmented matrix $(A \mid I)$ into $(I \mid A^{-1})$ by means of Gauss-Jordan elimination. Similarly, if A has an LU-decomposition, applying Gaussian elimination to A amounts to multiplying A from the left by the lower triangular matrix L^{-1}, resulting in $U = L^{-1}A$. Hence, the upper triangular matrix U is obtained without applying any *pivoting* steps. Putting $\det A = \det U = \prod_{i=1}^{d} u_{i,i}$ yields the determinant.

We will perform Gaussian elimination over finite fields of large prime order p, and we will do so for essentially uniformly random matrices only. As a consequence, there will be no need for pivoting and all computations will be exact. Inspired by Bareiss [16], we will combine division-free Gaussian elimination with back substitution such that $\det A$ is obtained at almost no extra cost. See Section 4.7.3 for further details.

4.4 Ridge Regression

Ridge regression is a well-known technique in statistics and machine learning [68], which can be seen as a refinement of the ordinary least squares method used in linear regression. Ridge regression provides the user with a handle, the *regularization parameter* $\lambda > 0$, that can be used to reduce the variance of the prediction at the cost of introducing some bias. If λ is set properly, ridge regression can outperform the ordinary least squares method in terms of the root mean-square error, defined below. In high-dimensional problems, ridge regression can help to reduce the problem of overfitting.

Given an overdetermined linear system $X\boldsymbol{w} = \boldsymbol{y}$, the least squares solution $\boldsymbol{w}$ minimizes $\|X\boldsymbol{w} - \boldsymbol{y}\|_2$. Typically, X is an $n \times d$ matrix over $\mathbb{R}$ with $n \gg d$. Each row of X represents an input record with d features, and the corresponding entry of $\boldsymbol{y}$ represents the known output value. The least squares solution $\boldsymbol{w} = (X^\mathsf{T}X)^{-1}X^\mathsf{T}\boldsymbol{y}$, is used as the optimal weight vector for predicting the output values for new input records $\boldsymbol{x}$ by evaluating $\boldsymbol{x}^\mathsf{T}\boldsymbol{w}$.

Ridge regression finds a vector $\boldsymbol{w}$ minimizing

$$\|X\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \lambda\|\boldsymbol{w}\|_2^2, \tag{4.1}$$

where we note the presence of the regularization parameter in the second term. The solution $\boldsymbol{w}$ minimizing (4.1) is now given by:

$$\boldsymbol{w} = \left(X^\mathsf{T}X + \lambda I\right)^{-1} X^\mathsf{T}\boldsymbol{y}. \tag{4.2}$$

To compute $\boldsymbol{w}$, one solves the linear system $A\boldsymbol{w} = \boldsymbol{b}$ with $A = X^\mathsf{T}X + \lambda I$ and $\boldsymbol{b} = X^\mathsf{T}\boldsymbol{y}$. Note that the regularization parameter λ not only suppresses large entries in $\boldsymbol{w}$, but also ensures that A is positive definite, hence invertible: $\boldsymbol{z}^\mathsf{T}\left(X^\mathsf{T}X + \lambda I\right)\boldsymbol{z} = \|X\boldsymbol{z}\|_2^2 + \lambda\|\boldsymbol{z}\|_2^2 > 0$ for any nonzero $\boldsymbol{z}$, since $\lambda > 0$.

In the context of machine learning, the input records X along with the known output values $\boldsymbol{y}$ are called the training set, and the least-squares solution $\boldsymbol{w}$ is called the model. The performance of the model is evaluated in terms of the root-mean-square error (RMSE) of the model's predictions. The model complexity (training error) is defined as the RMSE for the training set, which is equal to $\|X\boldsymbol{w} - \boldsymbol{y}\|_2/\sqrt{n}$. The generalizability (test error) of the model is defined as the RMSE for a test set $(X', \boldsymbol{y}')$, which is equal to $\|X'\boldsymbol{w} - \boldsymbol{y}'\|_2/\sqrt{n'}$. Overall, the goal is to ensure that both RMSEs are small and approximately equal to each other.

The performance of a machine learning algorithm critically depends on the quality of the input data. Extensive data preprocessing may be required in practice to enhance the quality of the input data. In our experiments we will use standard datasets from the UCI repository, for which most of the data preprocessing has already been done. The only two tasks that remain before applying ridge regression to these datasets are (i) feature scaling and (ii) encoding of categorical features.

For feature scaling, we apply min-max scaling to each of the columns of X and to vector $\boldsymbol{y}$ as well. Concretely, all features are scaled to the range $[-1,1]$. We prefer this form of data normalization because it requires little processing and does not leak too much information about X and $\boldsymbol{y}$.

To encode categorical features (including Boolean features), we basically use a form of "one-hot encoding" with respect to the range $[-1,1]$. For Boolean features, we encode the values True and False by 1 and -1, respectively. A categorical feature with s possible values is encoded by s Boolean features, where the value for exactly one of the Boolean features will be set to 1 and the remaining $s-1$ Boolean features are set to -1.

4.5 MPC Setting

We consider an MPC setting with m parties tolerating a dishonest minority of up to t passively corrupt parties, $0 \leq t \leq (m-1)/2$. The basic protocols for secure addition and multiplication over a finite field rely on Shamir secret sharing [19, 74]. For our practical experiments we use the MPyC framework [123], which succeeds the VIFF framework [72].

Let $p > m$ be a prime. We use $[a]_p$ or $[a]$ to denote a secret-shared value $a \in \mathbb{Z}_p$, where a is interpreted as a signed integer in the range $\{-\lfloor p/2 \rfloor, \ldots, \lfloor p/2 \rfloor\}$. We assume that secure field arithmetic $(+,-,*,/$ modulo $p)$ is supported efficiently as well as secure generation of random numbers (e.g., $[r]$ with $r \in_R \mathbb{Z}_p$).

We highlight three auxiliary protocols which are of particular relevance for our approach.

For secure dot products $[\boldsymbol{x}] \cdot [\boldsymbol{y}]$ with $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{Z}_p^d$, we recall that the complexity is the same as for a single secure multiplication, except for local computations. This extends to secure matrix products $[A][B]$ with $A, B \in \mathbb{Z}_p^{d \times d}$, for which the complexity is equivalent to d^2 secure multiplications in parallel.

Next, to generate $[r]$ and $[1/r]$ for a random $r \in_R \mathbb{Z}_p^*$, one proceeds as follows: generate $[r]$, $[u]$ with $r, u \in_R \mathbb{Z}_p$, open $[r][u]$ to obtain ru, and output $[r]$ and $[1/r] = [u]/(ru)$. For large p, $ru \neq 0$ will hold with overwhelming probability; if $ru = 0$ simply try again with fresh r and u.

Finally, we will also use secure conversion between secret-shared values in different prime fields. In particular, for primes p and q satisfying $p > q > 2^{\kappa+\ell}$, where κ is a security parameter, we use a secure protocol for converting $[a]_q$ into $[a]_p$, $-2^{\ell-1} \leq a < 2^{\ell-1}$. Roughly, such a protocol proceeds by generating $[r]_q$ and $[r]_p$ for a random $r \in [0, 2^{\ell+\kappa})$. Then the value of $a+r$ is opened, which is statistically indistinguishable from random for κ sufficiently large, and one sets $[a]_p = a + r - [r]_p$.

4.6 Solving Systems of Linear Equations

As outlined in Section 4.4, we divide ridge regression into two main stages. In the first stage we compute $A = X^{\mathsf{T}}X + \lambda I$ and $b = X^{\mathsf{T}}y$, and in the second stage we solve $A\boldsymbol{w} = \boldsymbol{b}$ to find $\boldsymbol{w}$. For secure ridge regression, the most interesting and challenging part will be the second stage, and in this section we motivate our approach for solving $A\boldsymbol{w} = \boldsymbol{b}$.

In numerical analysis one distinguishes two major types of solution methods for systems of linear equations. Direct methods, such as Gaussian elimination, run in a finite number of steps and compute an exact solution in the absence of rounding errors. Iterative methods, such as conjugate gradient, yield approximate solutions within a limited amount of time, even for very large matrices. In contrast to some other recent work (e.g., [71]), we will choose a direct method to solve $A\boldsymbol{w} = \boldsymbol{b}$ in our protocols for secure ridge regression. Below we explain our reasons for doing so.

An important observation is that we can actually use exact computation for secure ridge regression. Instead of relying on fixed-point arithmetic—or even worse floating-point arithmetic—in an MPC setting, we will only use exact integer arithmetic in our protocols. This way we take advantage of the fact that secure integer arithmetic is efficient even for large values when using Shamir secret sharing. Moreover, we can borrow techniques from the related setting of secure linear algebra over finite fields [46].

We will make sure that the input data (contained in X and $\boldsymbol{y}$) are scaled to integer values, basically by multiplying each input value with 2^α and rounding to the nearest integer for a fixed value of α. The value of α must be sufficiently large to ensure that the final results will be accurate. We will refer to α as the *accuracy* parameter.

Since A is invertible, solving $A\boldsymbol{w} = \boldsymbol{b}$ is equivalent to computing $\boldsymbol{w} = A^{-1}\boldsymbol{b}$. Therefore, even if A contains integer values only, the solution $\boldsymbol{w}$ will in general contain rational values. As $A^{-1} = (\det A)^{-1} \operatorname{adj} A$, however, it suffices to compute $\boldsymbol{w}' = (\operatorname{adj} A)\boldsymbol{b}$ and $z = \det A$, from which $\boldsymbol{w}$ can be recovered as $\boldsymbol{w} = \boldsymbol{w}'/z$. Here, both $\boldsymbol{w}'$ and z are integral. We compute $\boldsymbol{w}'$ and z by first reducing the augmented matrix $(A \mid b)$ to echelon form using Gaussian elimination and then applying back substitution to recover $\boldsymbol{w}$.

To perform *secure* Gaussian elimination on $(A \mid b)$ there are several options. A first idea is to use Gaussian elimination (row reduction) directly, which amounts to repeatedly selecting a pivot and updating the matrix accordingly. However, oblivious row reduction, hiding the position of the pivot and so on, is computationally very costly: searching for a nonzero element in the pivot column is already nontrivial in a secure setting, and obliviously swapping entire rows to move the pivot to the diagonal is even much more costly.

A common technique in numerical analysis to avoid pivot selection is the use of *preconditioning*. Roughly, the idea is to solve the equivalent system $RA\boldsymbol{w} = R\boldsymbol{b}$ for a random matrix R, instead of the original system $A\boldsymbol{w} = \boldsymbol{b}$. Matrix R is assumed to be invertible, which is true with overwhelming probability in many settings. When solving linear systems over $\mathbb{R}$, such an approach is numerically unstable and leads to poor results. When solving linear systems over a finite field, however, numerical instability is of no concern. We will follow this approach.

The upshot of computing $(\operatorname{adj} A)\boldsymbol{b}$ and $\det A$ separately is that we will also avoid the use of rational reconstruction. In the next section we will show why this lets us essentially *halve* the size of the prime modulus for the finite field arithmetic compared to other papers. For instance, [75] relies on rational reconstruction and uses a modulus which should be large enough to “hold” the *product* of $(\operatorname{adj} A)\boldsymbol{b}$ and $\det A$.

4.7 Secure Linear Algebra

We present protocols for computing determinants, matrix inverses, and solutions to linear systems. Given an invertible matrix $A \in \mathbb{Z}^{d\times d}$, we compute the results over $\mathbb{Z}_p$ assuming p is sufficiently large. E.g., for $-p/2 < \det A < p/2$, $\det A \in \mathbb{Z}_p^*$ and A is properly embedded in $\mathbb{Z}_p^{d\times d}$. Assuming further bounds on the entries of A and $\boldsymbol{b}$, we will show how to compute A^{-1} and $A^{-1}\boldsymbol{b}$ over $\mathbb{Z}_p$ as well.

```
Protocol 1 Det([A])                                      $A \in \mathrm{GL}_d(\mathbb{Z}_p)$
1: Generate $[R]$, $[\det R^{-1}]$ with $R \in_R \mathrm{LU}_d(\mathbb{Z}_p)$ using Protocol 2.
2: Open $RA \leftarrow [R][A]$.
3: Compute $[\det A] = \det(RA)[\det R^{-1}]$.
4: Return $[\det A]$.
```

4.7.1 Secure Determinant

Cramer and Damgård presented a protocol for secure computation of $\det A$ over any finite field [46], which is reminiscent of Bar-Ilan and Beaver's protocol for secure multiplicative inverses [14]. The idea is to securely generate a random invertible matrix $[R]$ together with its determinant $[\det R]$, open the randomized matrix RA, and finally compute $[\det A]$. We follow the same approach in Protocol 1, except that we improve upon the way random matrix R is generated in several ways.

Ideally, R is generated as a random matrix in $\mathrm{GL}_d(\mathbb{Z}_p)$. To securely compute $\det R$ as well, matrix R is limited to the slightly smaller range $\mathrm{LU}_d(\mathbb{Z}_p)$ of matrices that have an LU-decomposition. The following lemma shows that uniformly random matrices in $\mathrm{LU}_d(\mathbb{Z}_p)$ are statistically indistinguishable from uniformly random matrices in $\mathrm{GL}_d(\mathbb{Z}_p)$. Therefore, opening RA reveals negligible information on A only.

Lemma 2. *$\Delta(R;G) \le d/p$, for $R \in_R \mathrm{LU}_d(\mathbb{Z}_p)$ and $G \in_R \mathrm{GL}_d(\mathbb{Z}_p)$.*

Proof. Since $\mathrm{LU}_d(\mathbb{Z}_p) \subseteq \mathrm{GL}_d(\mathbb{Z}_p)$ and R and G are both uniform we have

$$\Delta(R;G) = \tfrac{1}{2} \sum_{x \in \mathrm{GL}_d(\mathbb{Z}_p)} |\Pr R = x - \Pr G = x| = 1 - \frac{|\mathrm{LU}_d(\mathbb{Z}_p)|}{|\mathrm{GL}_d(\mathbb{Z}_p)|}.$$

Since $|\mathrm{LU}_d(\mathbb{Z}_p)| = p^{d^2-d}(p-1)^d$ and $|\mathrm{GL}_d(\mathbb{Z}_p)| \le p^{d^2}$, we have

$$\Delta(R;G) \le 1 - \left(\frac{p-1}{p}\right)^d = 1 - \left(1 - \frac{1}{p}\right)^d \le \frac{d}{p},$$

using Bernoulli's inequality in the last step. □

To sample a matrix R securely from $\mathrm{LU}_d(\mathbb{Z}_p)$, we use Protocol 2. The protocol also outputs the determinant of R, or rather its inverse. Random matrices in $\mathrm{L}^1_d(\mathbb{Z}_p)$ and $\mathrm{U}_d(\mathbb{Z}_p)$ can be generated easily, provided we can securely generate random elements of $\mathbb{Z}_p$. To ensure that U is invertible, we generate $u_{i,i} \in_R \mathbb{Z}_p$ for $i = 1, \ldots, d$, and then apply secure inversion to $\det U = \prod_{i=1}^d u_{i,i}$. With negligible probability $\det U = 0$, in which case secure inversion will fail and we have to try again. With overwhelming probability, however, $\det U \ne 0$ and secure inversion will succeed. In total, Protocol 2 roughly uses d^2 random elements from $\mathbb{Z}_p$.

Our protocol for generating random matrices improves upon Cramer and Damgård's protocol Π_0 [46, p. 126] in several respects. The main difference is that protocol Π_0 depends on a redundant type of LU-decomposition in which the diagonals of both L and U consist of elements in $\mathbb{Z}_p^*$. By fixing the diagonal of L to all ones, the LU-decomposition used in our protocol is unique. As an immediate consequence, our proof for statistical indistinguishability is much simpler (cf. Lemma 2). Moreover, the complexity of the protocol is reduced as we do not need to generate the diagonal of L at random, and we do not need to compute $\det L$ either. Finally, as a further optimization, we only use one secure

Protocol 2 RndMatDet(d)

```
1: Generate $[L]$ with $L \in_R \mathrm{L}_d^1(\mathbb{Z}_p)$.
2: Generate $[U]$, $[\det U^{-1}]$ with $U \in_R \mathrm{U}_d(\mathbb{Z}_p)$.
3: Compute $[R] = [L][U]$.
4: Set $[\det R^{-1}] = [\det U^{-1}]$.
5: Return $[R]$, $[\det R^{-1}]$.
```

Protocol 3 AdjDet$([A])$ $A \in \mathrm{GL}_d(\mathbb{Z}_p)$

```
1: Generate $[R]$, $[\det R^{-1}]$ with $R \in_R \mathrm{LU}_d(\mathbb{Z}_p)$ using Protocol 2.
2: Open $RA \leftarrow [R][A]$.
3: Reduce $(RA \mid [R])$ to obtain $[A^{-1}]$ by Gauss-Jordan elimination over $\mathbb{Z}_p$.
4: Compute $[\det A] = \det(RA)[\det R^{-1}]$.
5: Compute $[\operatorname{adj} A] = [\det A][A^{-1}]$.
6: Return $[\operatorname{adj} A]$, $[\det A]$.
```

inversion throughout the entire protocol (to perform the secure zero-test and inversion for $\det U$ all at the same time).

Apart from generating a random matrix R and its inverse determinant, Protocol 1 mainly performs a secure matrix multiplication. The computation of $\det(RA)$ is done locally, so we might use any algorithm for computing determinants to implement this step. However, Lemma 2 helps us save some work for the local computation as well. The lemma basically implies that RA is statistically close to a uniformly random matrix in $\mathrm{LU}_d(\mathbb{Z}_p)$, and therefore we can perform Gaussian elimination to compute $\det(RA)$ without any pivoting, as shown below.

4.7.2 Secure Matrix Inversion

We next present Protocol 3 for secure matrix inversion, which is of independent interest. Since A^{-1} will in general have rational entries for a matrix $A \in \mathbb{Z}^{d\times d}$, as discussed above, we will use the pair $(\operatorname{adj} A, \det A)$ as representation of A^{-1}. This way we avoid any rational arithmetic, and, moreover, we can use a similar embedding for A in $\mathbb{Z}_p^{d\times d}$ as for the determinant, using the bound for $\|\operatorname{adj} A\|_{\max}$ from Section 4.3 to choose p sufficiently large.

If we stick to the common approach of computing $A^{-1} = (\det A)^{-1} \operatorname{adj} A$ over $\mathbb{Z}_p$, such that $\operatorname{adj} A$ and $\det A$ can be recovered using rational reconstruction over $\mathbb{Z}_p$, the required size for p would be roughly twice as large.

4.7.3 Secure Linear Solver

Finally, we present Protocol 4 for securely solving a linear system, in which we avoid performing a full matrix inversion. In step 3 we apply Gaussian elimination to the augmented matrix $(RA \mid [R][\boldsymbol{b}])$. As explained in Section 4.6, this can be done without pivoting. Matrix RA is first transformed into upper-triangular form, and then we apply back substitution to compute $[A^{-1}\boldsymbol{b}]$. For Gaussian elimination on $(RA \mid [R][\boldsymbol{b}])$, we use the division-free variant (see, e.g., [16]). Combined with back substitution, we achieve that $\det(RA)$ is obtained at almost no additional cost. In total we need $\frac{2}{3}d^3 + O(d^2)$ multiplications, $\frac{1}{3}d^3 + O(d^2)$ modular reductions, and exactly n inversions modulo p for step 3.

Protocol 4 LinSol($[A],[\boldsymbol{b}]$) $A \in \mathrm{GL}_d(\mathbb{Z}_p),\, \boldsymbol{b} \in \mathbb{Z}_p^d$

```
1: Generate $[R]$, $[\det R^{-1}]$ with $R \in_R \mathrm{LU}_d(\mathbb{Z}_p)$ using Protocol 2.
2: Open $RA \leftarrow [R][A]$.
3: Solve $(RA \mid [R][\boldsymbol{b}])$ to obtain $[A^{-1}\boldsymbol{b}]$ by Gaussian elimination over $\mathbb{Z}_p$.
4: Compute $[\det A] = \det(RA)[\det R]^{-1}$.
5: Compute $[(\mathrm{adj} A)\boldsymbol{b}] = [\det A][A^{-1}\boldsymbol{b}]$.
6: Return $[(\mathrm{adj} A)\boldsymbol{b}]$, $[\det A]$.
```

Protocol 5 Ridge($[X]_q,[\boldsymbol{y}]_q,\lambda$) $X \in \mathbb{Z}_q^{n\times d}, \boldsymbol{y} \in \mathbb{Z}_q^n, \lambda \in \mathbb{N}$

```
1: Compute $[A]_q = [X^\mathsf{T}]_q[X]_q + \lambda I$.                      $\triangleright A = X^\mathsf{T}X + \lambda I$
2: Compute $[\boldsymbol{b}]_q = [X^\mathsf{T}]_q[\boldsymbol{y}]_q$.                          $\triangleright \boldsymbol{b} = X^\mathsf{T}\boldsymbol{y}$
3: Convert $[(A \mid \boldsymbol{b})]_q$ to $[(A \mid \boldsymbol{b})]_p$.
4: Compute $([(\mathrm{adj} A)\boldsymbol{b}]_p, [\det A]_p) = \mathrm{LinSol}([(A \mid \boldsymbol{b})]_p)$.
5: Set $[(\det A)\boldsymbol{w}]_p = [(\mathrm{adj} A)\boldsymbol{b}]_p$.                    $\triangleright \boldsymbol{w} = A^{-1}\boldsymbol{b}$
6: Return $[(\det A)\boldsymbol{w}]_p, [\det A]_p$.
```

4.8 Secure Ridge Regression

In this section we present our protocol for ridge regression, see Protocol 5. All entries of X and $\boldsymbol{y}$ are assumed to be in $[-2^\alpha, 2^\alpha] \cap \mathbb{Z}$ for an appropriate value of the accuracy parameter α (i.e., normalized to $[-1,1]$ as explained in Section 4.4, scaled by a factor of 2^α, and rounded to the nearest integer). The regularization parameter λ is scaled accordingly. We note that parameter α is between 5 and 10 in our experiments, cf. Table 4.2.

The two main stages of ridge regression are performed over two different prime fields. In the first stage, $X^\mathsf{T}X + \lambda I$ and $X^\mathsf{T}\boldsymbol{y}$ are computed over a relatively small field $\mathbb{Z}_q$, while $\boldsymbol{w} = A^{-1}\boldsymbol{b}$ is computed over a substantially larger field $\mathbb{Z}_p$ in the second stage. See Table 4.2 for some typical sizes of p and q. Since n is typically very large as well, cf. Table 4.1, secure computation of $X^\mathsf{T}X$ over $\mathbb{Z}_p$ would put excessive demands on time and space utilization.

The conversion in step 3 of the protocol is done as described at the end of Section 4.5. The sizes for primes p and q are determined in the following lemma.

Lemma 3. *Let $\beta = \|(X \mid \boldsymbol{y})\|_{\max}$. Correctness of Protocol 5 follows if*

$$\frac{q}{2} > n\beta^2 + \lambda + 2^\kappa \quad \textit{and} \quad \frac{p}{2} > d(d-1)^{\frac{d-1}{2}}(n\beta^2+\lambda)^d.$$

Proof. For prime q we need that $q/2 > \|(A \mid \boldsymbol{b})\|_{\max}$. Each entry of $(A \mid \boldsymbol{b})$ is a dot product of two length-n vectors with entries bounded in absolute value by β, plus λ for the diagonal of A. Therefore, $\|(A \mid \boldsymbol{b})\|_{\max} \le n\beta^2 + \lambda$. To allow for secure conversion from $\mathbb{Z}_q$ to $\mathbb{Z}_p$, we require $q/2 > n\beta^2 + \lambda + 2^\kappa$.

For prime p we need that $p/2 > \det A$ and $p/2 > \|(\mathrm{adj} A)\boldsymbol{b}\|_\infty$. We use the bounds for $\det A$ and $\mathrm{adj} A$ obtained from Hadamard's inequality in Section 4.3 as follows. For the determinant of A, we have the bound $\det A \le (n\beta^2+\lambda)^d$ since A is symmetric positive definite. For the adjugate of A, we have the bound $\|(\mathrm{adj} A)\|_{\max} \le (d-1)^{\frac{d-1}{2}}(n\beta^2+\lambda)^{d-1}$. So, together with the bound $\|\boldsymbol{b}\|_\infty \le n\beta^2$, we take $p/2 > d(d-1)^{\frac{d-1}{2}}(n\beta^2+\lambda)^d$ as overall bound. □

id	dataset	n	d	target(s)	train RMSE	test RMSE
1	Student Performance	395	30	G3	0.38	0.46
2	Wine Quality red	1,599	11	quality	0.16	0.16
3	Wine Quality white	4,898	11	quality	0.19	0.19
4	Year Prediction MSD	515,345	90	year	0.21	0.21
5	Gas Sensor Array methane	4,178,504	16	ethylene methane	0.29 0.34	0.29 0.34
6	Gas Sensor Array CO	4,208,261	16	ethylene CO	0.34 0.34	0.34 0.34
7	HIGGS	11,000,000	7	class	0.97	0.97

Table 4.1: UCI datasets. RMSEs for ridge regression with Scikit-learn.

4.9 Performance Evaluation

We have performed several experiments using the UCI datasets [63] shown in Table 4.1. Each dataset is randomly split into a 70% training set and a 30% test set (except for dataset *Year prediction MSD*, for which we respect its predefined training/test split, taking the first 463715 rows to form the training set and the remaining 51630 rows are used for testing). The RMSEs reported for training and testing are obtained using the Cholesky solver provided for ridge regression in Scikit-learn [114], setting $\lambda = 1$. Note that the *Gas Sensor Array* datasets have two targets, for which the RMSEs are reported separately. To handle multiple targets, we have generalized Protocol 5 in the obvious way, replacing vector $\boldsymbol{y}$ by a matrix Y with one column per target.

We have run our protocol for secure ridge regression in a 3-party setting using the values for accuracy parameter α shown in Table 4.2. For each (normalized) dataset we have tried increasingly larger values for α until the errors became insignificant (below 0.1% relative to the RMSEs of Table 4.1). We have refrained from tuning the regularization parameter λ, and simply set $\lambda = 2^{2\alpha}$ (which corresponds to $\lambda = 1$ after scaling). The bit lengths $|p|$ and $|q|$ are determined from the bounds in Lemma 3, using $\beta = 2^{\alpha}$ and $\kappa = 30$. The total running time for Protocol 5 comprises two parts: $(A, \boldsymbol{b})$-time represents the time for computing $[A]$ and $[b]$ (steps 1–2), while $A^{-1}\boldsymbol{b}$-time covers the time for Protocol 4. The time for the conversion in step 3 is negligible.

Our implementation is written in Python using the MPyC package [123, ridgeregression.py]. The experiments were done using three PCs, connected via a Netgear GS208-100PES Ethernet switch. Each PC was running on Windows 8.1 Enterprise (64-bit) with an Intel Core i7-4770 CPU at 3.40GHz and 16GB of RAM. Table 4.2 compares our results to three other solutions for secure ridge regression from the literature. The times reported are purely indicative, and give a basic idea of the performance of the various solutions.

We note that the previous works shown in Table 4.2 exploit the locality of the input data, assuming that the data is either partitioned horizontally or vertically. For example, Nikolaenko et al. [110] assume the dataset is partitioned horizontally, allowing them to compute A and $\boldsymbol{b}$ using additive homomorphic encryption only. In our work, we do not make any specific assumptions on the distribution of the input data; any data provider simply sends secret shares of its data to the respective parties performing the secure computation (using $\log_2 q$ bits per share). Thus, in our experiments, each party holds shares of the *entire* dataset.

Also, these previous works [110, 71, 75] rely on a so-called 2-server approach requiring two non-colluding parties (e.g., a “crypto service provider” and an “evaluator”). For our solution, the number

				This work			[110]	[75]		[71]	
id	α	$\lvert q\rvert$	$\lvert p\rvert$	$(A, \boldsymbol{b})$	$A^{-1}\boldsymbol{b}$	total	HP	HP	VP	VP 32-bit	VP 64-bit
1	6	54	1316	0.13	1.48	1.61	-	39.76	328.06	5 (-0.0%)	35 (-0.0%)
2	7	58	314	0.02	0.08	0.10	39	-	-	-	-
3	8	61	357	0.04	0.12	0.16	45	4.09	-	0 (4.2%)	4 (-0.0%)
4	6	64	3105	237	18.3	255	-	-	-	230 (0.0%)	808 (0.0%)
5	8	71	675	62.8	0.05	62.9	-	-	-	-	-
6	9	73	709	63.0	0.05	63.1	-	-	-	42 (5.2%)	69 (0.0%)
7	5	66	277	34.9	0.12	35.0	-	-	-	-	-

Table 4.2: Results of this work compared to the literature. All times are in seconds. HP/VP stand for horizontal/vertical partitioning. 32-bit and 64-bit refer to bit lengths used for secure fixed-point arithmetic. Accuracy α yields relative errors below 0.1%. The relative errors reported by [71] are also given.

n	d	This work	[71]
50,000	20	1s	2s
50,000	100	24s	32s
500,000	20	11s	18s
500,000	100	3m29s	6m1s
1,000,000	100	6m57s	12m42s
1,000,000	200	28m30s	49m56s

Table 4.3: Comparison of $(A, \boldsymbol{b})$-times for synthetic data.

of colluding parties tolerated is scalable, assuming an honest majority.

The competitiveness of our solution is also confirmed by Table 4.3, showing our results for a range of synthetic datasets compared to the most favorable results reported by Gascón et al. [71] (for their 3-party setting).

4.10 Concluding Remarks

Assuming that matrix X is of full column rank, Protocol 5 can also be used for secure linear regression by setting $\lambda = 0$. If matrix X is distributed among several data providers, however, ensuring that X is of full rank need not be trivial. For instance, in a vertical data partitioning it may not that easy to detect a redundant feature (used by multiple data providers). Setting $\lambda > 0$ removes the need to remove such redundant columns.

Our results also extend to the underdetermined case $n < d$. In this case, the closed form solution given by Eq. (4.2) can be rewritten as

$$\boldsymbol{w} = X^{\mathsf{T}} \left(XX^{\mathsf{T}} + \lambda I \right)^{-1} \boldsymbol{y}, \tag{4.3}$$

```
Protocol 6 Ridge($[X]_q, [\boldsymbol{y}]_q, \lambda$)        $X \in \mathbb{Z}_q^{n\times d}, \boldsymbol{y} \in \mathbb{Z}_q^n, \lambda \in \mathbb{N}$
1: Compute $[A]_q = [X]_q[X^\top]_q + \lambda I$.                          $\triangleright A = XX^\top + \lambda I$
2: Convert $[(A \mid X \mid \boldsymbol{y})]_q$ to $[(A \mid X \mid \boldsymbol{y})]_p$.
3: Compute $([(\operatorname{adj} A)\boldsymbol{y}]_p, [\det A]_p) = \operatorname{LinSol}([(A \mid \boldsymbol{y})]_p)$.
4: Compute $[(\det A)\boldsymbol{w}]_p = [X^\top]_p[(\operatorname{adj} A)\boldsymbol{y}]_p$.            $\triangleright \boldsymbol{w} = X^\top A^{-1}\boldsymbol{y}$
5: Return $[(\det A)\boldsymbol{w}]_p, [\det A]_p$.
```

using that $(X^\top X + \lambda I)X^\top = X^\top XX^\top + \lambda X^\top = X^\top(XX^\top + \lambda I)$.

Modifying our protocol for ridge regression accordingly results in Protocol 6. In step 4 of the protocol the secret-shared matrix X converted to the large prime field $\mathbb{Z}_p$ is used to compute the output vector $\boldsymbol{w}$. Since typically $d \gg n$, a relatively small number of conversions n per entry of the length-d output vector $\boldsymbol{w}$ are performed. Setting $\lambda = 0$ for this protocol yields a solution for secure linear regression in the case that X is of full row rank.

The approach presented in this paper is generalized in follow-up work by Bouman and de Vreede [33], in which they show how to compute the Moore-Penrose pseudoinverse securely. Pseudoinverses are much harder to compute securely as one needs to hide all information about the rank of the input matrix.

Details about the handling of the input and output for our secure ridge protocols are beyond the scope of this paper. For instance, one needs to decide how much information the parties are willing to leak when normalizing their joint datasets. Also, the parties may jointly need to determine a suitable value for the regularization parameter λ (hyperparameter tuning). Similarly, the output $[(\det A)\boldsymbol{w}]_p, [\det A]_p$ of our secure ridge protocols can be handled in lots of ways. These two values may simply be revealed, accepting leakage of the exact value of the determinant. Alternatively, these values may be converted to shares over $\mathbb{Z}_{p'}$, where p' is of double length compared to p, followed by rational reconstruction modulo p' to recover $\boldsymbol{w}$ in the clear.

Chapter 5

Efficient Secure Computation with Silent Preprocessing

This chapter is based on the papers *Efficient Pseudorandom Correlation Generators: Silent OT Extension and More* [36] and *Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation* [35], co-authored by SODA researchers, published at IACR Crypto 2019 and ACM CCS 2019.

5.1 Introduction

There is a large body of work on optimizing the concrete efficiency of secure computation protocols via input-independent preprocessing. By securely generating many instances of simple correlations, one can dramatically reduce the online communication and computation costs of most existing protocols.

To give just one example, multiple independent instances of a random oblivious transfer[1] (OT) correlation can be used for secure two-party computation of Boolean circuits in the passive model, with communication cost of only two bits per party per (nonlinear) gate, and with computation cost that is comparable to computing the circuit with no security requirements at all [78]. Thus, assuming a fast communication network, protocols based on correlated randomness can achieve near-optimal performance.

The main challenge in applying this approach is the high concrete cost of securely generating the correlated randomness. Traditional solutions involve carefully optimized special-purpose secure computation protocols that have a high communication cost for each instance of the desired correlation [21, 56]. This holds even for the case of OT correlations, for which relatively fast OT extension techniques are known [85, 13, 94]. Moreover, even if offline communication is cheap, the cost of *storing* large amounts of correlated randomness for each party with whom a future interaction *might* take place can be significant.

Motivated by the limitations of traditional approaches for generating and storing correlated randomness, the notion of a *pseudorandom correlation generator* (PCG) was recently proposed and studied by Boyle et al. [34, 36]. The goal of a PCG is to compress long sources of correlated randomness without violating security. More concretely, a (two-party) PCG replaces a target two-party correlation, say many independent OT correlation instances, by a pair of *short* correlated keys, which

[1]In (a single instance of) a random OT correlation, one party obtains a pair of random bits (more generally, strings) (s_0, s_1) and the other obtains the pair (r, s_r) for a random bit r.

can be "silently" expanded *without any interaction*. The process of generating the correlated keys and locally expanding them should emulate an ideal process for generating the target correlation not only from the point of view of outsiders, but also from the point of view of *insiders* who can observe one of the two keys. Among other results, the aforementioned works of Boyle et al. [34, 36] obtain concretely efficient constructions of PCGs for OT correlations and vector oblivious linear evaluation (VOLE) correlations [87, 8] based on variants of the Learning Parity with Noise assumption [23]. These PCG constructions are motivated by the goal of secure computation with *silent preprocessing*, where a low-communication input-independent setup, followed by local ("silent") computation, enables a lightweight "non-cryptographic" online phase once the inputs are known.

However, towards realizing this goal, one major challenge remains: how can the pair of keys be securely generated? While the keys are short, their sampling algorithm is quite complex and involves multiple invocations of cryptographic primitives. Thus, even applying the fastest general-purpose protocols for generating these keys (e.g., optimized protocols based on garbled circuits [129]) incurs a very significant overhead.

An alternative approach for distributing the PCG key generation, suggested in [34, 36], relies on a recent special-purpose protocol of Doerner and shelat [60] for secure key generation of a distributed point function (DPF) [77, 37]. This protocol only makes a black-box use of symmetric cryptography and a small number of oblivious transfers, and hence it is also concretely efficient. Using this protocol for distributing the key generation of a PCG for OT correlations, Boyle et al. [36] obtained a *silent OT extension* protocol that generates (without any trusted setup) a large number of pseudo-random OTs from a small number of base OTs, using a low-communication setup followed by silent key expansion [36].

While the silent OT extension protocol from [36] and other protocols obtained using this approach have good concrete efficiency, they also have several limitations. First, they require a large number of communication rounds that grows (at least) logarithmically with the output length. Second, they are only secure against *passive* parties. Both of the above limitations are inherited from the DPF key generation protocol of [60]. Finally, their concrete efficiency estimates are not backed by an actual implementation, and ignore possible cache-misses and other system- and network-related sources of slowdown.

5.1.1 Our Contribution

In this work, we address the above limitations by making the following contributions.

Two-Round Silent OT Extension. We present the first *concretely efficient* two-round OT extension protocol, based on a variant of the LPN assumption. The protocol has a silent preprocessing feature, allowing the parties to push the bulk of the computational work to an offline phase. It can be used in two modes: either a random-input mode, where the communication complexity is sublinear in the output length, or a chosen-input mode, where the communication is roughly equal to the total input length. This applies even to the more challenging case of 1-bit OT, for which standard OT extension techniques that make a black-box use of symmetric cryptography [85, 13, 98, 94] have a high communication overhead compared to the input length. A key idea that underlies this improvement is replacing the DPF primitive in the PCG for OT from [36] by the simpler puncturable pseudorandom function (PPRF) primitive [97, 32, 38]. We design a parallel version of the distributed key generation protocol from [60] that applies to a PPRF instead of a DPF.

Our OT extension protocol bypasses a recent impossibility result of Garg et al. [70] on 2-round OT extension due to the use of the LPN assumption. While our construction (inevitably) does not fall into the standard black-box framework considered in [70], it still has a black-box flavor in that it only invokes a syndrome computation of *any* error-correcting code for which the LPN assumption holds. We remark that aside from its concrete efficiency, our 2-round OT extension protocol can be based on a conservative variant of (binary) LPN in a noise regime that is not known to imply public-key encryption, let alone oblivious transfer. Concretely, it can be instantiated by binary LPN in which the Hamming weight of the noise is higher than the $n^{1/2}$ bound required by the construction of Alekhnovich [5] and its variants.

The technique we use for generating OT correlations in two rounds can also be applied to VOLE correlations, as well as general protocols for non-interactive secure computation (NISC) with silent preprocessing.

Active Security. We present simple, practical techniques for secure distributed setup of PPRF keys with a weak form of active security. This suffices to upgrade our passive OT and VOLE protocols to active security, at a very low cost. Our main protocols in this setting have 4 rounds of interaction, but this can be reduced to 2 rounds using the Fiat-Shamir transform. We can also use this to obtain actively secure silent NISC or two-round OT extension on chosen inputs. These protocols are based on slightly stronger variants of LPN, where the adversary is allowed a single query to a one-bit leakage function on the error vector.

Implementation. We demonstrate the efficiency of our constructions with an implementation of our random OT extension protocol. The most costly part of the implementation is a large matrix-vector multiplication, which comes from applying the LPN assumption. We optimize this using a variant of LPN with quasi-cyclic codes, similarly to several recent, candidate post-quantum secure cryptosystems [9, 4], and present different tradeoffs with parameter choices. Our protocols have a very low communication overhead and perform significantly faster than previous, state-of-the-art protocols [85, 13, 94] in environments with restricted bandwidth. For instance, in a 100Mbps WAN setting, we are around 5x faster, and this improves to 47x in a 10MBps WAN. This is because, while our computational costs are around an order of magnitude higher, we need around 1000–2000 times less communication than the other protocols. We expect that additional optimizations of our implementation and the underlying error-correcting code will further improve the computational cost.

Applications. As well as the new application to NISC with silent preprocessing, our protocols can be applied to a range of traditional secure computation tasks. Below we mention just a few areas where we expect silent OT extension and VOLE to have an impact.

- *Semi-honest MPC for binary circuits.* In the passive "GMW protocol" [78], the correlated randomness needed to evaluate a Boolean circuit can be obtained from two random OTs per AND gate. Plugging in our random OT extension, we obtain a practical 2-PC protocol where each party communicates just 2 bits per AND gate on average. This is around 30x less communication than the state-of-the-art [59].

- *Malicious MPC for binary circuits.* Protocols based on authenticated garbling [129, 130] and BMR [81] are currently the state-of-the-art in actively secure MPC for binary circuits in a high-latency network. The main cost in these protocols comes from a preprocessing phase, where the parties use a large number of random, correlated oblivious transfers to produce correlated randomness. Our protocol can produce the same kind of oblivious transfers with almost zero communication, and we estimate this could reduce the *overall* communication in these protocols by around an order of magnitude.

- *Malicious MPC for arithmetic circuits.* The “SPDZ” family of protocols [21, 56, 54, 95, 96] uses information-theoretic MACs to achieve active security in MPC based on secret sharing. A large batch of these MACs can be created using a single instance of a long, random VOLE correlation, with essentially optimal communication. Plugging in our actively secure VOLE construction will reduce the costs of previous works that use either homomorphic encryption or string-OT to create MACs.

- *Private set intersection (PSI).* In circuit-based PSI, a generic 2-PC protocol is used to first compute a secret-sharing of the intersection of two sets, and then perform some useful computation on the result [82, 116, 117]. With the improvements to GMW mentioned above, we can expect to obtain a similar reduction in communication for these families of PSI protocols.

5.1.2 Technical Overview

We now give an overview of our silent constructions in the passive and active settings. For simplicity, we focus here on the case of 1-out-of-2 oblivious transfer.

We start by recalling the high-level idea of the pseudorandom correlation generators for vector-OLE (VOLE) and OT from [34, 36]. These constructions distribute a pair of seeds to a sender and a receiver, who can then locally expand the seeds to produce many instances of pseudorandom OT or VOLE. To do so, they use two main ingredients: a variant of the LPN assumption, and a method for the two parties to obtain a *compressed* form of random secret shares $\mathbf{v}_0, \mathbf{v}_1$, satisfying

$$\mathbf{v}_1 = \mathbf{v}_0 + \mathbf{e} \cdot x \in \mathbf{F}_{2^\lambda}^N \tag{5.1}$$

where $\mathbf{e} \in \{0,1\}^N$ is a random, sparse vector held by one party, and $x \in \mathbf{F}_{2^\lambda}$ is a random field element held by the other party.

Given this, the shares can be randomized by taking a public, binary matrix H that compresses from N down to $n < N$ elements, and locally multiplying each share with H. This works because $\mathbf{u} = \mathbf{e} \cdot H$ is pseudorandom under a suitable variant of LPN. Writing $\mathbf{v} = \mathbf{v}_0 \cdot H$ and $\mathbf{w} = \mathbf{v}_1 \cdot H$, from (5.1) we then get $\mathbf{w} = \mathbf{v} + \mathbf{u}x$. This can be seen as a set of random *correlated OTs*, where $u_i \in \{0,1\}$ are the receiver's choice bits, and $(v_i, v_i + x)$ are the sender's strings, of which the receiver learns w_i. These can be locally converted into random string-OTs with a standard hashing technique [85].

To obtain a compressed form of the shares in (5.1), the constructions of [34, 36] used a *distributed point function* (DPF) [77, 37]. Our first observation is that DPF is an overkill for this application,[2] and can be replaced with the simpler *puncturable pseudorandom function* (PPRF) primitive. A PPRF is a PRF F such that given an input x, and a PRF key k, one can generate a punctured key $\mathsf{k}\{x\}$ which allows evaluating F at every point except for x, and does not conceal any information about the value $F(\mathsf{k}, x)$. A PPRF can be built from any length-doubling pseudorandom generator, using a binary tree-based construction [97, 32, 38].

In the setup procedure, we will give the sender a random key k and x, and give to the receiver a random point $\alpha \in \{1, \ldots, N\}$, a punctured key $\mathsf{k}\{\alpha\}$, and the value $z = F(\mathsf{k}, \alpha) + x$. Given these seeds, the sender and receiver can now define the expanded outputs, for $i = 1, \ldots, n$:

[2] In contrast, we do not know how to replace DPF by PPRF in some of the other PCG constructions from [36], including the LPN-based constructions for low-degree correlations and the PRG-based constructions for one-time-truth-table correlations.

$$\mathbf{v}_0[i] = F(\mathsf{k}, i), \quad \mathbf{v}_1[i] = \begin{cases} F(\mathsf{k}, i) & i \neq \alpha \\ z & \text{otherwise} \end{cases}$$

These immediately satisfy (5.1), with **e** as the α-th unit vector. To obtain sharings of sparse **e** with, say, t non-zero coordinates, as needed to use LPN, we repeat this t times and XOR together all t sets of outputs.

Conceptually, this construction is simpler than using a DPF, and moreover, as we now show, it brings several efficiency advantages.

Two-Round Setup of Puncturable PRF Keys. We present a simple, two-round protocol for distributed the above setup with passive security, inspired by the DPF setup protocol of Doerner and shelat [60]. The core of our protocol is the following procedure. For each of t secret LPN noise coordinates $\alpha_j \in [N]$ known to the receiver, the sender generates a fresh PRF key k_j, and wishes to obliviously communicate a punctured key $\mathsf{k}_j\{\alpha_j\}$ and hardcoded punctured output $z_j = PRF(\mathsf{k}_j, \alpha) + x$ to the receiver. Combined, this yields a secret sharing of the vector $x \cdot \mathbf{e}$, as required. To do so, for each $\mathsf{k}\{\alpha\}$, the parties made use of $\ell = \log N$ parallel OT executions: the sender's ℓ message pairs correspond to appropriate sums of partial evaluations from a consistent GGM PRF tree and his secret value x, and the receiver's ℓ selection bits correspond to the bits of his chosen path α.

Compared with previous works based on distributed point functions [34, 36, 60], the number of rounds of interaction collapses from $O(\log N)$ to just two, given any two-round OT protocol. This is possible since the punctured point α is known to the receiver, whereas when α is secret-shared as in a DPF, the OTs in the setup procedure seem hard to parallelize.

Two-Round OT Extension and Silent NISC. We observe that in the two-round setup, the receiver can *already compute* part of its output before sending the first round message. In the case of OT, this part corresponds to its random vector of choice bits **u**. This means that the receiver can already *derandomize* its OT outputs in the first round, by sending in parallel with its setup message the value $\mathbf{u} + \mathbf{c}$, where **c** is its *chosen* input vector. Since the sender can compute its random OT outputs after the first round, this leads to a two-round OT extension protocol that additionally enjoys the "silent preprocessing" feature of pushing the bulk of the computation to an offline phase, before the inputs are known. This can be generalized from OT to VOLE and other useful instances of *non-interactive secure computation* (NISC) [86], simultaneously inheriting the silent preprocessing feature from the PCG and the minimal interaction feature from an underlying NISC protocol.

Actively Secure Setup. In the above passive setup procedure, a active *receiver* has no cheating space; altered selection bits merely correspond to a different choice of noise coordinate $\alpha' \in [N]$. However, a active *sender* may generate message pairs inconsistent with any correct PRF evaluation tree, or use inconsistent inputs x across the t executions (in which case the outputs are not valid shares of $x \cdot \mathbf{u}$ for any single x). For example, by injecting errors into one of the two messages within an OT message pair, the sender can effectively "guess" and learn a bit of α, and will go unnoticed if his guess is correct.

We demonstrate that with small overhead, we can restrict a active sender to *only* such selective-failure attacks. This is formalized via an ideal functionality where the adversarial sender can send a guess range $I \subseteq [N]$ for α, a "getting caught" predicate is tested as a function of the receiver's true input, and the functionality either aborts or delivers the output accordingly. We then show that paired

				LAN (10Gbps) times				WAN (100Mbps) times				WAN (10Mbps) times			
					n				n				n		
Protocol	Base type	λ	τ	10^7	10^6	10^5	10^4	10^7	10^6	10^5	10^4	10^7	10^6	10^5	10^4
This (SH)	hybrid	128	4	2,441	208	**76**	**67**	**2,726**	**513**	**422**	**425**	**2,756**	**518**	**454**	**422**
IKNP	base	128	4	**268**	**125**	94	91	13,728	1,850	493	459	128,954	13,332	1,756	445
This (SH)	hybrid	128	1	7,990	533	130	100	**8,252**	**808**	**451**	422	**8,291**	**815**	**467**	**422**
IKNP	base	128	1	**573**	**157**	**108**	**98**	15,622	2,030	613	**341**	129,011	13,285	1,672	429
This (Mal)	hybrid	128	4	2,659	280	**84**	**78**	**2,872**	**479**	**457**	**424**	**2,846**	**515**	**438**	**422**
KOS	base	128	4	**333**	**121**	110	111	13,722	1,933	589	426	129,052	13,391	1,804	536
This (Mal)	hybrid	128	1	8,765	584	141	**104**	**9,055**	**828**	**460**	**423**	**8,929**	**831**	**467**	**433**
KOS	base	128	1	**674**	**170**	**113**	106	15,741	2088	702	433	129,771	13,389	1,772	518

Figure 5.1: The running time in milliseconds of our implementation compared to [13] in both the LAN (0ms latency) and WAN (40ms one-way latency) settings, with security parameter $\lambda = 128$. λ is the computational security parameter. We set the scaling N/n to 2. τ denotes the number of threads. Hybrid refers to doing 128 base OTs followed by IKNP to derive the total required base OTs.

with an interactive leakage notion for LPN, this suffices to give us PCG setup protocols for VOLE and OT with active security.

Our basic actively secure protocols have 4 rounds, but this can be compressed to two rounds with the Fiat-Shamir transform, in the random oracle model. Just as in the passive protocols, we can convert the setup protocols into NISC protocols, this time under a slightly stronger variant of LPN with one bit of *adaptive* leakage on the error vector, obtaining two-round OT extension with active security.

5.2 Performance Results

We now present the performance results from our implementation of the new silent OT extension protocol. For full details of the protocols and further optimizations we used, beyond the above informal description, see the full paper [35].

5.2.1 Instantiating the Code and Parameters

The most costly part of our implementation is a matrix-vector multiplication with the public matrix H used in the dual-LPN assumption. We optimize this by instantiating H using the parity-check matrix of a quasi-cyclic code. Multiplication by H can then be done with polynomial arithmetic in $\mathbb{Z}_2[X]/(X^n - 1)$, for which we use the library `bitpolymul` [43]. Another optimization that improves efficiency and reduces the seed size is to use a regular error distribution, where the error vector $\mathbf{e} \in \mathbf{F}_2^N$ is the concatenation of t random unit vectors, each of length N/t. To choose the code parameters N, n and the error weight t, we analyze security against the best known attacks, additionally accounting for a $\sqrt{N}$ speedup that can be obtained from the DOOM attack [124] when using quasi-cyclic codes. As also observed in [80], we are not aware of any attacks that exploit regular errors and perform significantly better than usual.

In the full paper, we provide more details on selecting parameters and describe some further optimizations for the syndrome computation.

5.2.2 Results

We implement our passive and active secure protocols and report their performance in several different settings. The source code can be found at `https://github.com/osu-crypto/libOTe`. The

		Total Comm. (bytes)				Comm./OT (bits)			
		n				n			
Protocol	Base type	10^7	10^6	10^5	10^4	10^7	10^6	10^5	10^4
This (SH/Mal)	hybrid	126,658	98,754	83,394	57,806	0.101	0.790	6.672	46.245
IKNP/KOS	base	160,056,360	16,011,518	1,655,784	168,186	128.045	128.092	132.463	134.549

Figure 5.2: The communication overhead of our implementation compared to [85, 94], with $N/n = 2$ and $\lambda = 4$. See Figure 5.1.

benchmark was performed on a single AWS c4.4xLarge instance with network latency artificially limited to emulate a LAN or WAN settings. Specifically, we consider a LAN setting with bandwidth of 10Gbps and 0ms latency and two WAN settings with 100, 10 Mbps & 40ms one-way latency. We compare with the passive OT extension protocol of Ishai et al. [85] (IKNP) and the active secure protocol of [94] (KOS) as implemented by a state-of-the-art library. Both our implementations and that of [85, 94] use the same three round active secure base OT protocol of Naor & Pinkas[108]. We note that our protocols can be composed with a two round base OT protocol to give a two round OT extension. In the WAN setting this optimization would reduce the running times by approximately 40ms for all protocols.

The functionality we realize is to produce $n \in \{10^4, 10^5, 10^6, 10^7\}$ uniformly random OTs of length 128 bits. One distinction between our protocol and [85, 94] is that the choice bits of the receiver are uniformly chosen by our protocol, while [85, 94] allows the receiver to specify them. These random OTs can then be de-randomized with additional communication.

Figure 5.1 contains the running time of our protocol. A fuller table, with alternative choices of parameters (security parameter λ, scaling parameter N/n, method for computing the base OTs) is available in the full version. The primary takeaway is that both of our protocols achieve extremely low communication while the total running time remains competitive with or superior to KOS and IKNP. We report running times with each party having 1 or 4 threads, along with a background IO thread. In the LAN setting with sub-millisecond latency & 10Gbps we observe that the IKNP and KOS protocols achieve significant performance, requiring just 0.26 or 0.33 seconds to compute 10 million OTs with a single thread. While the computational cost of IKNP and KOS does outperform our implementation by roughly one order of magnitude, it also requires between 1000 and 2000 times more communication. This difference means that for more realistic network settings, such as 100Mbps, our implementation achieves a faster running time. With 4 threads and a limit of 100Mbps our implementation is up to 5 times faster (counting total running time, including both local computation and communication costs) and remains faster even for small n where our communication overheads are asymptotically closer together.

For the constrained setting of 10Mbps our protocol truly stands out with a 47 times speedup compared to IKNP with $n = 10^7$ and $t = 4$. We see a similar 46 times speedup in the active setting compared to KOS. Moreover, when comparing between the across the different network settings our protocol incurs minimal to no perform impact from decreasing bandwidth. For instance, with a 10Gbps connection our passive protocol processes $n = 10^7$ OTs in 2.4 seconds while with 1000 times less bandwidth the protocol still just requires 2.8 seconds.

This scalability is explained in Figure 5.2 which contains the communication overhead of our protocol. A fuller table, with alternative choices of parameters (security parameter λ, scaling parameter N/n, method for computing the base OTs) is available in the full version. We parameterize our protocols by the desired security level $\lambda \in \{80, 128\}$ and a tunable parameter $s = N/n$. The latter controls a trade-off between the number of PPRF evaluations and length of the resulting vectors. To maintain security level of λ bits, increasing s results in fewer PPRF evaluations and less communication. However, it also increases the computational overhead. Our smallest running times were achieved with $s = 2$. However, we also consider $s = 4$ which decreases our total communication from 126KB to 80KB for $n = 10^7$. In contrast, the IKNP protocol requires 160MB for the same security level. This represents as much as a 2000 times reduction in communication. This low communication overhead results in our protocol requiring as little as 0.038 bits per OT for $n = 10^7$ and $\lambda = 80$. In our worst case of $n = 10^4$ our protocol still requires between 3 and 6 times less communication than IKNP. Another compelling property of our protocol is that we incur near constant additive communication overhead when comparing our active and passive protocols.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, and others. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Mark Abspoel, Niek J. Bouman, Berry Schoenmakers, and Niels de Vreede. Fast secure comparison for medium-sized integers and its application in binarized neural networks. In Mitsuru Matsui, editor, *CT-RSA 2019*, volume 11405 of *LNCS*, pages 453–472. Springer, Heidelberg, March 2019.

[3] 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. Cryptology ePrint Archive, Report 2019/872, 2019. https://eprint.iacr.org/2019/872.

[4] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, and Gilles Zémor. Hamming quasi-cyclic (HQC). 2019.

[5] Michael Alekhnovich. More on average case vs approximation complexity. In *44th FOCS*, pages 298–307. IEEE Computer Society Press, October 2003.

[6] Alexandra Institute. FRESCO - a FRamework for Efficient Secure COmputation. https://github.com/aicis/fresco.

[7] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. MCMix: Anonymous messaging via secure multiparty computation. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1217–1234, Vancouver, BC, 2017. USENIX Association.

[8] Benny Applebaum, Ivan Damgård, Yuval Ishai, Michael Nielsen, and Lior Zichron. Secure arithmetic computation with constant computational overhead. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 223–254. Springer, Heidelberg, August 2017.

[9] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, et al. Bike: Bit flipping key encapsulation. 2019.

[10] 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.

[11] 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.

[12] David W. Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. From keys to databases - real-world applications of secure multi-party computation. *The Computer Journal*, 61(12):1749–1771, 2018.

[13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 535–548. ACM Press, November 2013.

[14] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In *Proc. 8th Symp. on Princip. of Distr. Comp.*, pages 201–209, NY, 1989. ACM.

[15] Assi Barak, Daniel Escudero, Anders Dalskov, and Marcel Keller. Secure evaluation of quantized neural networks. Cryptology ePrint Archive, Report 2019/131, 2019. https://eprint.iacr.org/2019/131.

[16] Erwin H. Bareiss. Sylvester's identity and multistep integer-preserving gaussian elimination. *Mathematics of Computation*, 22(103):565–578, 1968.

[17] 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.

[18] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[19] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proc. 20th Symposium on Theory of Computing (STOC '88)*, pages 1–10, New York, 1988. ACM.

[20] 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.

[21] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.

[22] Frank Blom, Niek J. Bouman, Berry Schoenmakers, and Niels de Vreede. Efficient secure ridge regression from randomized gaussian elimination. Cryptology ePrint Archive, Report 2019/773, 2019. https://eprint.iacr.org/2019/773.

[23] Avrim Blum, Merrick L. Furst, Michael J. Kearns, and Richard J. Lipton. Cryptographic primitives based on hard learning problems. In *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, pages 278–291, 1993.

[24] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013.

[25] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In Rainer Böhme and Tatsuaki Okamoto, editors, *FC 2015*, volume 8975 of *LNCS*, pages 227–234. Springer, Heidelberg, January 2015.

[26] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and Taxes: a Privacy-Preserving Study Using Secure Computation. *PoPETs*, 2016(3):117–135, 2016.

[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.

[28] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Secur.*, 11(6):403–418, November 2012.

[29] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis - (short paper). In Angelos D. Keromytis, editor, *FC 2012*, volume 7397 of *LNCS*, pages 57–64. Springer, Heidelberg, February / March 2012.

[30] 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.

[31] 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.

[32] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Heidelberg, December 2013.

[33] Niek J. Bouman and Niels de Vreede. A practical approach to the secure computation of the Moore-Penrose pseudoinverse over the rationals. Cryptology ePrint Archive, Report 2019/470, 2019.

[34] Elette Boyle, Geoffroy Couteau, Niv Gilboa, and Yuval Ishai. Compressing vector OLE. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 896–912. ACM Press, October 2018.

[35] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round ot extension and silent non-interactive secure computation. In *ACM CCS*, 2019.

[36] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, August 2019.

[37] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

[38] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, Heidelberg, March 2014.

[39] Dario Catalano, Mario Di Raimondo, Dario Fiore, and Irene Giacomelli. Monza: Fast maliciously secure two party computation on z_{2^k}. Cryptology ePrint Archive, Report 2019/211, 2019. https://eprint.iacr.org/2019/211.

[40] 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.

[41] 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.

[42] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. Astra: High throughput 3pc over rings with application to secure prediction. Cryptology ePrint Archive, Report 2019/429, 2019. https://eprint.iacr.org/2019/429.

[43] Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Multiplying boolean polynomials with frobenius partitions in additive fast fourier transform. *CoRR*, abs/1803.11301, 2018.

[44] 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.

[45] M. De Cock, R. Dowsley, C. Horst, R. Katti, A. Nascimento, W. Poon, and S. Truex. Efficient and private scoring of decision trees, support vector machines and logistic regression models based on pre-computation. *IEEE Transactions on Dependable and Secure Computing*, pages 1–1, 2018.

[46] Ronald Cramer and Ivan Damgård. Secure distributed linear algebra in a constant number of rounds. In *Proc. CRYPTO 2001, Santa Barbara, USA*, pages 119–136. Springer, 2001.

[47] 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.

[48] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 342–362. Springer, Heidelberg, February 2005.

[49] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. In Jens Grossklags and Bart Preneel, editors, *FC 2016*, volume 9603 of *LNCS*, pages 169–187. Springer, Heidelberg, February 2016.

[50] 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 *IEEE Security & Privacy*, 2019.

[51] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 285–304. Springer, Heidelberg, March 2006.

[52] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 160–179. Springer, Heidelberg, March 2009.

[53] Ivan Damgård and Marcel Keller. Secure multiparty AES. In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 367–374. Springer, Heidelberg, January 2010.

[54] 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.

[55] 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.

[56] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

[57] Ivan Damgård and Rasmus Winther Zakarias. Fast oblivious AES A dedicated application of the MiniMac protocol. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 16*, volume 9646 of *LNCS*, pages 245–264. Springer, Heidelberg, April 2016.

[58] 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.

[59] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the communication barrier in secure computation using lookup tables. In *NDSS 2017*. The Internet Society, February / March 2017.

[60] Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 523–535. ACM Press, October / November 2017.

[61] Nico Döttling, Satrajit Ghosh, Jesper Buus Nielsen, Tobias Nilges, and Roberto Trifiletti. TinyOLE: Efficient actively secure two-party computation from oblivious linear function evaluation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2263–2276. ACM Press, October / November 2017.

[62] Wenliang Du, Yunghsiang S Han, and Shigang Chen. Privacy-preserving multivariate statistical analysis: Linear regression and classification. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 222–233. SIAM, 2004.

[63] Dheeru Dua and Casey Graff. UCI machine learning repository, 2019.

[64] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! arithmetic 3pc for any modulus with active security. Cryptology ePrint Archive, Report 2019/164, 2019. https://eprint.iacr.org/2019/164.

[65] Hendrik Eerikson, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! arithmetic 3pc for any modulus with active security. *IACR Cryptology ePrint Archive*, 2019:164, 2019.

[66] 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.

[67] Matthias Fitzi, Nicolas Gisin, Ueli M. Maurer, and Oliver von Rotz. Unconditional byzantine agreement and multi-party computation secure against dishonest minorities from scratch. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 482–501. Springer, Heidelberg, April / May 2002.

[68] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[69] 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.

[70] Sanjam Garg, Mohammad Mahmoody, Daniel Masny, and Izaak Meckler. On the round complexity of OT extension. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 545–574. Springer, Heidelberg, August 2018.

[71] Adrià Gascón, Phillipp Schoppmann, Borja Balle, Mariana Raykova, Jack Doerner, Samee Zahur, and David Evans. Privacy-preserving distributed linear regression on high-dimensional data. *Proceedings on Privacy Enhancing Technologies*, 2017(4):345–364, 2017.

[72] M. Geisler. *Cryptographic Protocols: Theory and Implementation*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, February 2010. viff.dk.

[73] 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.

[74] R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *17th annual ACM symposium on Principles of Distributed Computing (PODC '98)*, pages 101–111, New York, 1998. ACM.

[75] Irene Giacomelli, Somesh Jha, Marc Joye, C. David Page, and Kyonghwan Yoon. Privacy-preserving ridge regression with only linearly-homomorphic encryption. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security - 16th International Conference, ACNS 2018, Leuven, Belgium, July 2-4, 2018, Proceedings*, volume 10892 of *Lecture Notes in Computer Science*, pages 243–261. Springer, 2018.

[76] 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 *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 201–210, 2016.

[77] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, Heidelberg, May 2014.

[78] 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.

[79] Rob Hall, Stephen E. Fienberg, and Yuval Nardi. Secure multiple linear regression based on homomorphic encryption, 2011.

[80] Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. TinyKeys: A new approach to efficient multi-party computation. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 3–33. Springer, Heidelberg, August 2018.

[81] Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 598–628. Springer, Heidelberg, December 2017.

[82] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. The Internet Society, February 2012.

[83] 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.

[84] 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.

[85] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[86] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. Efficient non-interactive secure computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 406–425. Springer, Heidelberg, May 2011.

[87] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 294–314. Springer, Heidelberg, March 2009.

[88] 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.

[89] Thomas P. Jakobsen, Jesper Buus Nielsen, and Claudio Orlandi. A framework for outsourcing of secure computation. In *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014*, pages 81–92, 2014.

[90] Marc Joye and Fariborz Salehi. Private yet efficient decision tree evaluation. In Florian Kerschbaum and Stefano Paraboschi, editors, *Data and Applications Security and Privacy XXXII - IFIP WG*, volume 10980 of *Lecture Notes in Computer Science*, pages 243–259. Springer, 2018.

[91] 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.

[92] Jonathan Katz, Samuel Ranellucci, Mike Rosulek, and Xiao Wang. Optimizing authenticated garbling for faster secure two-party computation. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 365–391. Springer, Heidelberg, August 2018.

[93] Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Eduardo Soria-Vazquez, and Srinivas Vivek. Faster secure multi-party computation of AES and DES using lookup tables. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 229–249. Springer, Heidelberg, July 2017.

[94] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Heidelberg, August 2015.

[95] 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.

[96] 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.

[97] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 669–684. ACM Press, November 2013.

[98] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 54–70. Springer, Heidelberg, August 2013.

[99] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images, 2009. https://www.cs.toronto.edu/~kriz/cifar.html.

[100] 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.

[101] 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.

[102] Helger Lipmaa and Tomas Toft. Secure equality and greater-than tests with sublinear online complexity. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *ICALP 2013, Part II*, volume 7966 of *LNCS*, pages 645–656. Springer, Heidelberg, July 2013.

[103] 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.

[104] Eleftheria Makri, Dragos Rotaru, Nigel P. Smart, and Frederik Vercauteren. EPIC: Efficient private image classification (or: Learning from the masters). In Mitsuru Matsui, editor, *CT-RSA 2019*, volume 11405 of *LNCS*, pages 473–492. Springer, Heidelberg, March 2019.

[105] 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.

[106] 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.

[107] N1 Analytics. MP-SPDZ - Versatile framework for multi-party computation. https://github.com/n1analytics/MP-SPDZ.

[108] Moni Naor and Benny Pinkas. Computationally secure oblivious transfer. *Journal of Cryptology*, 18(1):1–35, January 2005.

[109] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[110] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 334–348, Washington, DC, USA, 2013. IEEE Computer Society.

[111] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *PKC 2007*, volume 4450 of *LNCS*, pages 343–360. Springer, Heidelberg, April 2007.

[112] Claudio Orlandi, Alessandro Piva, and Mauro Barni. Oblivious neural network computing via homomorphic encryption. *EURASIP Journal on Information Security*, 2007(1):037343, 2007.

[113] 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.

[114] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[115] Martin Pettai and Peeter Laud. Automatic proofs of privacy of secure multi-party computation protocols against active adversaries. In Cédric Fournet, Michael W. Hicks, and Luca Viganò, editors, *IEEE Computer Security Foundations Symposium*, pages 75–89. IEEE Computer Society, 2015.

[116] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 515–530. USENIX Association, August 2015.

[117] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based psi with linear communication. 2019. `https://eprint.iacr.org/2019/241`.

[118] Ariadna Quattoni and Antonio Torralba. Recognizing indoor scenes. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 413–420. IEEE Computer Society, 2009.

[119] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In *21st ACM STOC*, pages 73–85. ACM Press, May 1989.

[120] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin Lauter, and Farinaz Koushanfar. Xonn: Xnor-based oblivious deep neural network inference. Cryptology ePrint Archive, Report 2019/171, 2019. `https://eprint.iacr.org/2019/171`.

[121] 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.

[122] 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, 2018.

[123] Berry Schoenmakers. MPyC: Secure multiparty computation in Python. GitHub, May 2018. github.com/lschoe/mpyc.

[124] Nicolas Sendrier. Decoding one out of many. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*, pages 51–67. Springer, Heidelberg, November / December 2011.

[125] 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.

[126] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826. IEEE Computer Society, 2016.

[127] University of Bristol. SPDZ-2. https://github.com/bristolcrypto/SPDZ-2.

[128] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: Efficient and private neural network training. Cryptology ePrint Archive, Report 2018/442, 2018. https://eprint.iacr.org/2018/442.

[129] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 21–37. ACM Press, October / November 2017.

[130] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 39–56. ACM Press, October / November 2017.

[131] Wikipedia. Lee sedol. https://en.wikipedia.org/wiki/Lee_Sedol. Accessed: 19-12-2018.

[132] David J. Wu, Tony Feng, Michael Naehrig, and Kristin E. Lauter. Privately evaluating decision trees and random forests. *PoPETs*, 2016(4):335–355, 2016.