MACHINE LEARNING ON ENCRYPTED DATA

INAUGURALDISSERTATION
ZUR ERLANGUNG DES AKADEMISCHEN GRADES
EINES DOKTORS DER NATURWISSENSCHAFTEN
DER UNIVERSITÄT MANNHEIM

vorgelegt von

Dipl.-Math. Angela Wiesberg geb. Jäschke
aus Berlin

Mannheim, 2018

Dekan: Dr. Bernd Lübcke

Referent: Prof. Dr. Frederik Armknecht, Universität Mannheim

Korreferent: Prof. Dr. Colin Boyd, NTNU Trondheim

Tag des Fachvortrags: 31.08.2018

ABSTRACT

In a time in which computing power has never been cheaper and the possibilities of extracting knowledge from data seem ever-increasing, the idea of doing this while protecting the user's privacy seems too good to be true. However, with the introduction of the first Fully Homomorphic Encryption scheme in 2009, we now have at our disposal a whole collection of encryption schemes that allow arbitrary computations on encrypted data. With this primitive, a user can encrypt his data, send it somewhere to be analyzed, and obtain the encrypted result – all without divulging anything about the data to the computing party. This is especially useful in the context of Machine Learning: A service provider can have a model that returns predictions on input data, and a user can obtain these predictions on his data without having to share it with the service provider. This is particularly important because this data is often of a sensitive nature, e.g. in medical or financial contexts.

While Fully Homomorphic Encryption schemes solve this problem on a high level, there are some challenges in practice. A prominent one is the issue of encoding: Real-world data usually consists of rational numbers, whereas the plaintext space of Fully Homomorphic Encryption schemes is generally a finite field. Thus, we need an efficient way to encode the data into these plaintext spaces, and a guideline which of the finite fields to choose in the first place. Since Fully Homomorphic Encryption schemes are still very slow computationally, this choice has a huge impact on the performance. The efficiency of different encoding choices is measured with three metrics: The number of additions we need to perform in the underlying plaintext space for a given computation on the rational numbers, the number of multiplications in the plaintext space, and the multiplicative depth. The latter measures the number of consecutive multiplications in the plaintext space needed to perform a computation on rational numbers, and is motivated by the concrete structure of the Fully Homomorphic Encryption schemes we have today.

In this work, we first show in Chapter 2 that among all finite fields $GF(p^k)$, when adding or multiplying two natural numbers, the choice $GF(2)$ is best in terms of the number of additions and multiplications. In terms of multiplicative depth there is no generic optimum, as this depends on the concrete function and the input length of the function arguments. However, we do show that choosing $k > 1$ always has worse performance than choosing $GF(p)$.

Because of this finding, we focus on the encoding base $GF(2)$ in the rest of the work. In Chapter 3, we extend our analysis to include negative numbers, and thus examine the effort incurred by the two most popular encoding for signed numbers, Two's Complement and Sign-Magnitude. We see that Two's Complement is better for adding, and Sign-Magnitude is better for multiplying two numbers. We utilize this fact to invent a new encoding, called Hybrid Encoding, which essentially switches between the two to minimize the effort. Our new encoding induces a performance gain of over 70% in some of our applications.

We then extend our analysis from integer to rational numbers in Chapter 4. We propose several optimizations, which result in an efficiency gain of over 95%. We also propose ways to speed up the comparison function, and to reduce bitlengths in computations where some assumptions are met. The latter reduces the runtime by over 76% in our computations.

In Chapter 5, we apply our findings to algorithms from Machine Learning: We perform a classification using the Linear Means Classifier, and see the large impact that our Hybrid Encoding has. We then tackle the more complicated task of training a Machine Learning algorithm on encrypted data. We use the Perceptron for this, and see that our length management procedure can decrease runtimes enormously. We also again see that the Hybrid Encoding vastly outperforms the other two encodings. Lastly, we move to the clustering problem from the area of unsupervised learning, where we run the K-Means-Algorithm on encrypted data. We adapt the algorithm to make it efficiently executable in the Fully Homomorphic Encryption context, and show that the performance of this new algorithm is similar to that of the original K-Means-Algorithm in terms of clustering accuracy. The runtime is reduced by more than 95% compared to straightforward approaches of executing the K-Means-Algorithm on encrypted data.
We thus see that we can indeed perform algorithms from the world of Machine Learning on encrypted data, and that by choosing the encoding wisely and employing optimizations where we can, we can significantly speed up computations. We also see that modifying an algorithm to make it executable on encrypted data at all can yield results comparable to the original algorithm, and is thus a promising way to extend the class of algorithms we can evaluate on encrypted data.
This work is based on the publications [ABC+15], [JA16], [JA17] and [JA18].

ZUSAMMENFASSUNG

Wir leben in einer Zeit, in der Rechenleistung nie günstiger war und die Möglichkeiten, Wissen aus Daten zu gewinnen, stetig zunehmen. Die Idee, dies ohne Einschränkung der Privatsphäre des Nutzers zu tun, klingt hier fast zu gut, um wahr zu sein. Nach der Entdeckung des ersten vollhomomorphen Verschlüsselungsverfahrens im Jahr 2009 stehen uns allerdings inzwischen eine ganze Reihe von Verschlüsselungsverfahren zur Verfügung, die sogar beliebige Berechnungen auf verschlüsselten Daten erlauben. Mit diesem Ansatz kann ein Nutzer seine Daten verschlüsseln und versenden, um sie von einem Anbieter analysieren zu lassen, und erhält das verschlüsselte Resultat dieser Analyse zurück – alles, ohne dem Anbieter den Inhalt der Daten offenzulegen. Dies ist besonders im Kontext des maschinellen Lernens nützlich: Ein Service-Anbieter kann so über ein Modell verfügen, welches aus Eingabedaten Vorhersagen berechnet, und der Nutzer kann diese Vorhersagen für seine Daten erhalten, ohne selbige dem Anbieter offenlegen zu müssen. Dies ist vor allem dann wichtig, wenn die Daten sensibel sind, z.B. im medizinischen oder Finanz-Bereich.

Während vollhomomorphe Verschlüsselungsverfahren dieses Problem im Prinzip lösen, gibt es in der Praxis einige Hindernisse zu überwinden. Ein besonders wichtiger Aspekt ist hier die Kodierung: Die Daten bestehen meist aus rationalen Zahlen, wohingegen die Nachrichtenräume der vollhomomorphen Verschlüsselungsverfahren endliche Körper sind. Man benötigt also einen möglichst effizienten Weg, um die Daten in den Nachrichtenräumen zu kodieren. Ferner benötigt man eine Richtlinie, welchen endlichen Körper man überhaupt als Nachrichtenraum wählen sollte. Da vollhomomorphe Verschlüsselungsverfahren noch sehr langsam in ihren Berechnungen sind, hat diese Wahl einen enormen Einfluss auf ihre Performanz. Die Effizienz der verschiedenen Wahlmöglichkeiten der Kodierung wird anhand von drei Metriken gemessen: Erstens die Anzahl der Additionen, die im zugrundeliegenden Nachrichtenraum ausgeführt werden müssen, um eine gegebene Berechnung auf den rationalen Daten auszuführen. Zweitens die Anzahl der Multiplikationen im Nachrichtenraum, und drittens die multiplikative Tiefe. Letztere misst, wie viele aufeinanderfolgende Multiplikationen man im Nachrichtenraum ausführen muss, um eine Berechnung auf den rationalen Daten durchzuführen, und ist in der konkreten Struktur der heutigen vollhomomorphen Verschlüsselungsverfahren begründet.

In dieser Arbeit zeigen wir zunächst in Kapitel 2, dass die Addition oder Multiplikation zweier rationalen Zahlen am besten im endlichen Körper $GF(2)$ geschehen sollte, wenn man die Anzahl der Additionen oder Multiplikationen im Nachrichtenraum als Metrik betrachtet. Wählt man die mutliplikative Tiefe als Metrik, so existiert kein generisches Optimum, da die geringste Tiefe von der konkreten Funktion, die berechnet werden soll, und der Länge der Eingaben dieser Funktion abhängt. Wir zeigen jedoch, dass die Wahl $k > 1$ für die endlichen Körper $GF(p^k)$ niemals von Vorteil ist, und man stattdessen stets mit endlichen Körpern der Form $GF(p)$ arbeiten sollte.

Aufgrund dieser Resultate beschränken wir uns im Rest dieser Arbeit auf den Nachrichtenraum $GF(2)$. In Kapitel 3 erweitern wir unsere Analyse auf negative Zahlen und vergleichen zu diesem Zweck die Kosten für die zwei beliebtesten Kodierungen für ganze Zahlen, Two's Complement und Sign-Magnitude. Wir stellen fest, dass Two's Complement besser

ist, um zwei Zahlen zu addieren, und dass Sign-Magnitude besser ist, um sie zu multiplizieren. Wir nutzen diese Tatsache, um eine neue Kodierung zu entwickeln, welche wir Hybrid Encoding nennen. Sie wechselt im Wesentlichen zwischen Two's Complement und Sign-Magnitude, um die Kosten zu minimieren. Unsere neue Kodierung reduziert für einige unserer Anwendungen die Laufzeit um mehr als 70%.
Wir erweitern anschließend in Kapitel 4 unsere Analyse von den ganzen auf die rationalen Zahlen. Wir stellen verschiedene Optimierungen vor, welche die Effizienz um mehr als 95% steigern. Wir zeigen außerdem, wie man den Vergleich von zwei verschlüsselten Zahlen beschleunigen kann und wie man die Bitlänge unter bestimmten Annahmen verkleinern kann. Letzteres reduziert die Laufzeit unserer Berechnungen um über 76%.
Schließlich wenden wir in Kapitel 5 unsere Erkenntnisse auf Algorithmen aus dem Bereich des maschinellen Lernens an: Wir führen zunächst eine Klassifizierung mit dem Linear Means Classifier durch und sehen, welch großen Einfluss unser Hybrid Encoding hierbei hat. Anschließend nehmen wir uns die kompliziertere Aufgabe vor, ein Modell auf verschlüsselten Daten zu trainieren. Wir verwenden hierzu das Perceptron und stellen fest, dass unsere Reduzierung der Bitlängen die Laufzeit enorm verkürzen kann. Wir sehen zudem auch hier wieder, dass unser Hybrid Encoding deutlich schneller als die anderen beiden Kodierungen ist. Schließlich wenden wir uns dem Clustering-Problem aus dem Gebiet des unüberwachten Lernens zu und implementieren den K-Means-Algorithmus auf verschlüsselten Daten. Wir adaptieren den Algorithmus, um ihn im Kontext der vollhomomorphen Verschlüsselung effizient ausführbar zu machen, und zeigen, dass die Genauigkeit des neuen Algorithmus vergleichbar mit der des ursprünglichen K-Means-Algorithmus ist. Die Laufzeit wird hierdurch um mehr als 95% reduziert im Vergleich zu trivialen Ansätzen, den K-Means-Algorithmus auf verschlüsselten Daten auszuführen.
Wir schließen also, dass wir Algorithmen aus dem Bereich des maschinellen Lernens in der Tat auf verschlüsselten Daten ausführen und so die Privatsphäre des Nutzers schützen können. Außerdem sehen wir, dass eine geschickte Wahl der Kodierung und die Anwendung verschiedener Optimierungen die Effizienz erheblich verbessern. Wir stellen auch fest, dass die Modifizierung eines bestehenden Algorithmus mit dem Ziel, ihn auf verschlüsselten Daten ausführbar zu machen, zu einer mit dem ursprünglichen Algorithmus vergleichbaren Genauigkeit führen kann. Somit können wir die Klasse derjenigen Algorithmen, welche wir auf verschlüsselten Daten ausführen können, erheblich erweitern.
Diese Arbeit basiert auf den Veröffentlichungen [ABC+15], [JA16], [JA17] und [JA18].

ACKNOWLEDGMENTS

First and foremost, I would like to thank my husband Stefan Wiesberg. You always provided a much-needed balance to my life, and without your constant love and support I could not have come this far. I am truly grateful, and I am excited for all the adventures that our future together holds.
I would also like to thank my advisor, Frederik Armknecht, for guiding me through my years as a PhD student and creating an environment that made me feel valued and respected. Your feedback helped me improve, not just in the context of this thesis, yet I still had much freedom to find my academic style. You created many professional opportunities for me, and were also always supportive in private matters.
Further, I want to thank my family for being there for and supporting me all these years. You always believed in me and encouraged me to pursue my dreams and ambitions, and it makes me very happy that we share such a strong bond.
I also want to thank all of my colleagues not just for the many fun times we had, but for their productive input as well. While my gratitude goes out to all of you, I want to thank in particular Matthias Hamann and Christian Müller for their invaluable help in getting things like the encryption libraries to run, and Christian Gorke for the great companionship – our coffee and lunch breaks are one of the things I will miss most when I leave the university.
Additionally, I would like to thank Matthias Krause: I learned a great deal not only from the classes we taught together, but also from the countless conversations we had about anything and everything, which I truly enjoyed.
Last but not least, I want to thank Colin Boyd not only for agreeing to be the second advisor on my thesis, but also for the fruitful collaboration we had and especially the great trip to Norway that resulted from it.

Machine Learning on Encrypted Data

List of Figures

List of Tables

List of Algorithms

Chapter 1

INTRODUCTION

Fully Homomorphic Encryption (FHE) describes a class of encryption schemes that in principle allow arbitrary computations on encrypted data. This makes them a promising tool for outsourcing sensitive data without compromising privacy. Originally proposed in 1978 in [RAD78], it wasn't until 2009 that the first scheme achieving this property was published in [Gen09]. This initial scheme was highly inefficient and more a proof of concept than anything else, but many further schemes followed, and efficiency is slowly approaching real-world feasibility.
Concretely, assume that there is a party $\mathcal{A}$ who has some confidential data that they would like to outsource to party $\mathcal{B}$, perhaps in a cloud setting. If $\mathcal{A}$ wants to compute some function on this outsourced data, like a sum over certain values, or the retrieval of a subset of the data, they would conventionally have two choices:

1. The data is outsourced in unencrypted form. This means that $\mathcal{B}$ has fully flexibility in computing the function for $\mathcal{A}$ and returning only the result, but $\mathcal{B}$ also has full access to the confidential data, thus violating privacy.

2. The data is encrypted before outsourcing it. This maintains data privacy by hiding the contents from $\mathcal{B}$, but $\mathcal{B}$ cannot do anything with the data – if $\mathcal{A}$ wants to compute a function on the data, they must retrieve all the data, decrypt it, and perform the computation themselves.

We see that conventionally, there is a tradeoff between functionality and privacy. Fully Homomorphic Encryption, however, offers a solution that guarantees privacy while maintaining functionality: Here, $\mathcal{A}$ can encrypt the data and outsource the ciphertexts to $\mathcal{B}$. When $\mathcal{A}$ requests a function of their data, $\mathcal{B}$ can compute this function directly on the ciphertexts, obtaining the result of the calculation in encrypted form. This result is sent back to $\mathcal{A}$, who can decrypt it with the secret decryption key. Since $\mathcal{B}$ only saw encrypted data, no information about the underlying confidential data is revealed (except metadata, which is unavoidable), and $\mathcal{A}$ did not have to download their entire database just to perform a single computation.
Of course, in practice there are many issues that need to be addressed. A prominent one, which makes up a large part of this work, is the question of encoding: Most data and functions that $\mathcal{A}$ may want to apply to that data operate over rational numbers, whereas

the plaintext spaces of FHE schemes are usually finite fields. Thus, one needs to map the input numbers into appropriate plaintext spaces in the most efficient way possible - a problem we solve in Chapters 2 to 4. Alternatively, another approach is to modify the underlying algorithm that $\mathcal{A}$ wants to apply to the data so that the resulting algorithm is more FHE-friendly and can be more easily applied by $\mathcal{B}$ – we explore this idea in the context of Machine Learning in Section 5.
The rest of this introduction is structured as follows: Section 1.1 showcases the importance of FHE by presenting some potential applications. Section 1.2 gives a formal definition of FHE, explains the differences to related notions, and gives an overview of existing schemes and implementations. Section 1.3 explains the difficulties that arise in practice and motivates the contribution of this work.
A large part of the contents of this chapter has been published in [ABC+15].

1.1 APPLICATIONS OF FULLY HOMOMORPHIC ENCRYPTION

In this section, we present some potential applications of FHE to show the magnitude of the impact this area of research could have when the involved algorithms become sufficiently efficient.

1.1.1 Standalone Applications

We first examine some scenarios for which Fully Homomorphic Encryption offers a straightforward solution, before presenting more complex structure which use FHE as a building block in Section 1.1.2.

Consumer Privacy in Advertising

Even though advertising has a negative connotation for most people, it can actually be a useful enhancement if tailored to a user's needs and preferences, e.g. via recommender systems or location-based advertising. However, many people are worried about their privacy: Real-time location sharing essentially allows live tracking of the user, and consumer preferences may divulge information that the user is not comfortable sharing. Fully Homomorphic Encryption could be used in this scenario by encrypting the user's location or preferences, and having the tailored advertisement be the encrypted result, which is only decrypted locally by the user and thus hidden from the service provider. Examples of this application area are [JPH13], which presents a recommender system that operates via a social network and chooses products based on (confidential) tastes of a users friends, [AS11], where a user can get content from a recommender system without the system knowing which content was sent, and [NLV11], where a location-based advertising system (e.g. for discount vouchers of nearby shops) with encrypted location is theoretically conceived. The latter requires the advertisements to come from a third party which does not collude with the service provider.

Medical Applications

Among the most privacy-critical data associated with any individual is his medical data. In order to utilize current knowledge, e.g. to predict diseases or monitor general health

from things like blood pressure, heart rate, weight or blood sugar readings, a user must divulge his medical records so that they can be fed into the respective algorithms. By doing this in encrypted form, the patient's data remains confidential, and the algorithm is only applied to the ciphertexts. The user can decrypt the result with his secret key, and nobody else sees the original data or the result. This approach was described in [NLV11], and works like [LLN14] and [KL15] examine privacy-preserving genome analysis using Homomorphic Encryption.

Financial Privacy

In [NLV11], it is proposed that a company which has sensitive data and a proprietary algorithm, e.g. a stock portfolio and a stock price prediction algorithm in the financial sector, could upload both in encrypted form in order to outsource computations. However, Fully Homomorphic Encryption does not guarantee that it is possible to encrypt the function, i.e., the prediction algorithm in this case. What it does offer is that party $\mathcal{A}$ may have sensitive information like a stock portfolio, and party $\mathcal{B}$ may own a proprietary algorithm like a stock price prediction algorithm. Then $\mathcal{A}$ can send their encrypted data to $\mathcal{B}$, who runs the algorithm on the ciphertexts and returns the encrypted result. Fully Homomorphic Encryption guarantees that $\mathcal{B}$ did not learn anything about $\mathcal{A}$'s data, and if the encryption scheme has a property called *circuit privacy* (see Section 1.2.1), it also guarantees that $\mathcal{A}$ learns nothing about $\mathcal{B}$'s algorithm except the result on the input data. Thus, $\mathcal{A}$ may get $\mathcal{B}$'s prediction on their portfolio without either party having to disclose their information.

Forensic Image Recognition

The goal in forensic image recognition is to detect illegal images in data sets. This can be done by storing the hash values of forbidden images and comparing them to the hash values of the data set entries. One major concern regarding this approach is that perpetrators could obtain the hash database, check if their images match any hash values, and if so, modify them slightly so that they are no longer detected. In [BPHJ14], a solution to this problem is proposed: The hash database could be encrypted, and the check is performed on the hashed and encrypted database so that the result is also encrypted and can only be decrypted by the database owner (e.g., the police).

Machine Learning

Machine Learning is a field that focuses on extracting information from data. When applied to very large datasets, it is often referred to as Data Mining. Generally, there are two approaches in Machine Learning: Supervised and unsupervised. In supervised learning, there is a training set with known classifications, and the goal is to build a model that will be able to classify new, previously unseen instances. Both the training and the classification of new data can be done on encrypted data with FHE. This application was first proposed in [GLN12], and since then it has been a popular area of research. Many publications choose (Deep) Neural Networks (see [GDL+16], [CdWM+17], [PAH+17], [BPTG15] and our publication, [JA16]) and (Linear) Regression or Hyperplane Classification (e.g., [LKS16], [EAH17], [BSS+17] or [GLN12]) as their supervised learning algorithms, though there is also some work on other algorithm classes like decision trees and random forests

in [WFNL16], or logistic regression e.g. in [BLN14], [KSW+18], or [KSK+18]. A more detailed overview of works concerning Machine Learning on encrypted data will be given in Section 5.1.1.3. For the area of unsupervised learning, our publication [JA18] is to our knowledge the only work concerned with this application of FHE. Our results in applying Fully Homomorphic Encryption to this fascinating area of research will be presented in Chapter 5.

1.1.2 FHE as a Building Block

We now present some advanced concepts which could be realized using FHE as a building block.

Zero Knowledge Proofs

In his PhD thesis [Gen09], Gentry shows how to construct a non-interactive zero knowledge (NIZK) proof from Fully Homomorphic Encryption: The challenge is to prove knowledge of a satisfying assignment of input bits to a Boolean circuit without revealing the inputs. To do this, the prover encrypts the input bits with FHE and evaluates the circuit on the ciphertexts, which yields the encrypted output bit (which should encrypt 1 if the prover indeed posesses the knowledge he claims to have). A standard NIZK proof showing that each input ciphertext encrypted either 0 or 1, and that the output encrypts 1, then suffices to prove the claim to the verifier.

Verifiable Computation

In cloud computing, the two main functionalities are data storage and outsourcing computations. We have already motivated FHE as a means to keep the data private while still allowing the user to outsource his computations – however, Fully Homomorphic Encryption can be useful even when the data is not sensitive and can be stored in the clear: If the cloud service only returns the result, the user has no way of being sure that the result is actually correct, as the service may be faulty or even economically motivated to cheat on costly computations. Fully Homomorphic Encryption can be used as a tool that allows the user to verify correctness of the computation, for example through homomorphic Message Authentication Codes (MACs). Each input has a tag that was encrypted with FHE by the data owner, and the party doing the computation can combine these tags to obtain one which authenticates the result of the computation. It must hold that verifying the correctness is much more efficient than performing the computation on the data and only possible with the secret key of the data owner. Additionally, the output tag should be independent of the size of the dataset. Publications in this field of research include [CKV10], [GGP10] and [GW13].

Signatures

Homomorphic signatures can be considered as a public key-version of the homomorphic MACS mentioned in the previous paragraph. The idea is again that the input data has been signed with the data owner's private key, and the computing party can compute a function of the input data and a valid signature for the result of this computation, using the input signatures and FHE. Anybody can then verify the correctness of the signed

output with the public key (this is the difference to homomorphic MACs, where only the owner of the secret key can verify the signature). Fully homomorphic signatures were first defined in [BF11], and [GVW15] continues this line of work.

Multiparty Computation

Multiparty computation (MPC) is a primitive that is often seen as a rival to FHE in a sense, as use cases for both approaches are very similar. MPC is usually faster, but requires interaction in order to perform computations, so the choice depends on the concrete scenario. More details on MPC can also be found in Section 1.2.2.3. However, in some cases, a hybrid approach could have benefits: For example, in [DPSZ12] the authors propose a way of using FHE[1] to perform a costly multiplication in an offline preprocessing phase, and switch back to MPC for the main computation.

1.2 FHE OVERVIEW

In this section, we will give an introduction to Fully Homomorphic Encryption. We will first discuss some formal definitions, then we will present an overview of existing work in the FHE realm, and lastly we will distinguish FHE from other, closely related notions.

1.2.1 Formal Definition

We now present a series of definitions that culminate in the definition of an FHE scheme. Note that originally, these notions were defined only for the binary case, i.e., when the plaintext space of the encryption scheme is $\{0, 1\}$. We have generalized the definitions from [ABC+15] to the more general setting of rings as plaintext spaces to accommodate all current FHE schemes. This was mainly done by replacing the notion of circuits with polynomials.
We denote the following spaces:

- $\mathcal{P}$ is the plaintext space.
- $\mathcal{X}$ is the space of fresh encryptions (the image of the encryption algorithm Enc).
- $\mathcal{Y}$ is the space of valid evaluation outputs (the image of the evaluation algorithm Eval).
- $\mathcal{Z} = \mathcal{X} \cup \mathcal{Y}$ is the space of ciphertexts that are fresh, or valid evaluation outputs.
- $\mathcal{K}_p, \mathcal{K}_s$, and $\mathcal{K}_e$ are keyspaces for pk, sk, and evk. The public key contains a description of the plaintext and ciphertext spaces.
- $\mathbb{P}$ is the set of *permitted polynomials*, i.e. all the polynomials which the scheme can evaluate.

[1] To be exact, they use the related notion *Somewhat Homomorphic Encryption*, which will be explained in Section 1.2.2.

Definition 1.1 ($\mathbb{P}$-Evaluation Scheme[2])**.** A $\mathbb{P}$-evaluation scheme for a set $\mathbb{P}$ of multivariate polynomials is a tuple of probabilistic polynomial–time algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ such that:

$\mathsf{Gen}(1^\lambda, \alpha)$ is the key generation algorithm. It takes two inputs, the security parameter λ and an auxiliary input α, and outputs a key triple (pk, sk, evk), where pk is the key used for encryption, sk is the key used for decryption and evk is the key used for evaluation.

$\mathsf{Enc}(pk, m)$ is the encryption algorithm. As input it takes the encryption key $pk \in \mathcal{K}_p$ and a plaintext $m \in \mathcal{P}$. Its output is a ciphertext $c \in \mathcal{X}$.

$\mathsf{Eval}(evk, p, c_1, \ldots, c_n)$ is the evaluation algorithm. It takes as inputs the evaluation key $evk \in \mathcal{K}_e$, a polynomial $p \in \mathbb{P}$ and a tuple of inputs that can be a mix of fresh ciphertexts and previous evaluation results. It produces an evaluation output.

$\mathsf{Dec}(sk, c)$ is the decryption algorithm. It takes as input the decryption key $sk \in \mathcal{K}_s$ and either a ciphertext or an evaluation output and produces a plaintext $m \in \mathcal{P}$.

The reason that we differentiate between fresh encryptions and outputs of the evaluation algorithm is that these can look very different. In fact, it is actually not required that a fresh ciphertext be decryptable, only that evaluation outputs are. If one wanted to decrypt a fresh ciphertext, one could run Eval on it with the polynomial $p(x) = x$ to obtain an evaluation output which is decryptable. In other words, if the decryption algorithm does not accept fresh ciphertexts, we can replace it by an algorithm that runs all fresh ciphertexts through the identity function to obtain an evaluation output, and then decrypts the result with the original decryption function. This results in a decryption algorithm that can decrypt both fresh ciphertexts and evaluation outputs – i.e., it operates on the whole space $\mathcal{Z} = \mathcal{X} \cup \mathcal{Y}$. Due to this workaround, we ignore this technicality and assume all ciphertexts are decryptable.
We now move on to a series of attributes that are required of a $\mathbb{P}$-evaluation scheme in order for it to be called fully homomorphic.

Definition 1.2 (Correct Decryption)**.** A $\mathbb{P}$-evaluation scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ is said to *correctly decrypt* if for all $m \in \mathcal{P}$,

$$\Pr[\mathsf{Dec}(sk, \mathsf{Enc}(pk, m)) = m] = 1,$$

where sk and pk are outputs of $\mathsf{Gen}(1^\lambda, \alpha)$.

This means that we must be able to correctly decrypt ciphertexts – an attribute always required of encryption schemes, homomorphic or not. Since we are focusing on homomorphic schemes, we would of course also like to require that computations on encrypted data return the correct result as well: Evaluating a function on ciphertexts and decrypting the result should yield the same value as as the corresponding computation on the unencrypted input data. Due to the structure of all existing schemes, it is not impossible for this requirement to fail from time to time, so we soften the definition by allowing failure with negligible[3] probability.

[2] Definition loosely based on [BV11a].

[3] A function $f(x)$ is called negligible if for every positive polynomial $p(x)$ there exists an x_p such that for all $x > x_p$, it holds that $|f(x)| \leq 1/p(x)$. This means that the absolute value of f decreases faster than the inverse of any polynomial.

Definition 1.3 (Correct Evaluation[4])**.** A $\mathbb{P}$-evaluation scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ *correctly evaluates* all polynomials in $\mathbb{P}$ if for every $c_1, \ldots, c_n \in \mathcal{X}$, where $m_i \leftarrow \mathsf{Dec}(sk, c_i)$, for every $p \in \mathbb{P}$, and some negligible function ε,

$$\Pr[\mathsf{Dec}(sk, \mathsf{Eval}(evk, p, c_1, \ldots, c_n)) = p(m_1, \ldots, m_n)] = 1 - \varepsilon(\lambda)$$

where sk, pk and evk are outputs of $\mathsf{Gen}(1^\lambda, \alpha)$.

Definition 1.4 (Correctness)**.** A $\mathbb{P}$-evaluation scheme is *correct* if it correctly decrypts and evaluates all polynomials in $\mathbb{P}$.

It may seem like requiring correctness would be enough to define a homomorphic encryption scheme, but it allows a trivial construction that can evaluate any function and fulfills these definitions: Take any regular encryption scheme, define Eval to be the identity function (i.e., $\mathsf{Eval}(evk, p, c_1, \ldots, c_n) = (p, c_1, \ldots, c_n)$) and redefine the decryption function to decrypt the ciphertexts $c_1, \ldots, c_n$, apply the polynomial p to theses plaintexts, and return the result of this computation. Obviously, this construction does not satisfy what we mean by homomorphic encryption, so we need further requirements to exclude this trivial scheme. We do this by restricting the size that the output of Eval may have.

Definition 1.5 (Compactness[5])**.** A $\mathbb{P}$-evaluation scheme is called *compact* if there is a polynomial f such that for any key-triple (sk, pk, evk) output by $\mathsf{Gen}(1^\lambda, \alpha)$, any polynomial $p \in \mathbb{P}$ and any ciphertexts $c_1, \ldots, c_n \in \mathcal{X}$, the size of the output $\mathsf{Eval}(evk, p, c_1, \ldots, c_n)$ is not more than $f(\lambda)$ bits, independent of the size of the polynomial p.

Joining this definition of compactness with our requirement for correctness, we obtain the notion of *compact evaluation*:

Definition 1.6 (Compact Evaluation)**.** A $\mathbb{P}$-evaluation scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ *compactly evaluates* all polynomials in $\mathbb{P}$ if the scheme is compact and correct.

Before we define Fully Homomorphic Encryption, we present one last auxiliary definition. It is not technically required for FHE, but it plays an important role for our use case in Chapter 5, and nearly all modern schemes achieve it. It is called *Circuit Privacy* because it was first defined in [Gen09], which had a binary plaintext space and thus utilized circuits instead of polynomials, but we have adapted it to our setting. Intuitively, it captures the notion that an output of Eval should not reveal anything about the function that was applied except what can be derived from the decrypted result.

Definition 1.7 (Perfect/Statistical/Computational Circuit Privacy[6])**.** A $\mathbb{P}$-evaluation scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$ is said to be *perfectly/statistically/computationally circuit private* if for any key-triple (sk, pk, evk) output by $\mathsf{Gen}(1^\lambda, \alpha)$, for all polynomials $p \in \mathbb{P}$ and all $c_i \in \mathcal{X}$, such that $m_i \leftarrow \mathsf{Dec}(sk, c_i)$, the two distributions on $\mathcal{Z}$

$$D_1 = \mathsf{Eval}(evk, p, c_1, \ldots, c_n)$$

and

$$D_2 = \mathsf{Enc}(pk, p(m_1, \ldots, m_n)),$$

both taken over the randomness of each algorithm, are *perfectly indistinguishable*, *statistically indistinguishable* or *computationally indistinguishable*, respectively.

[4] Definition based on [BV11a], Def. 3.3.
[5] Definition based on [vDGHV10], Def. 3.
[6] Definition based on [Gen09], Def. 2.1.6.

There is also a slightly weaker, but less used notion called *function privacy* which merely requires that the outputs of evaluating different functions on encrypted data are perfectly/ statistically /computationally indistinguishable.
We can now define what it means that an encryption scheme is *fully homomorphic*.

Definition 1.8 (Fully Homomorphic Encryption Scheme[7])**.** A $\mathbb{P}$-evaluation scheme is said to be *fully homomorphic* if it is compact, correct, and the set of permitted polynomials $\mathbb{P}$ is the set of **all** polynomials.

This means that the scheme must be able to evaluate any polynomial in a compact fashion correctly, and the polynomial does not have to be known at encryption time.
Finally, we present one definition that is motivated from the concrete schemes we have today rather than a theoretical necessity for Fully Homomorphic Encryption. We present it at this point because it plays a large role for the choices made in the rest of this work.

Definition 1.9 (Bootstrapping)**.** A $\mathbb{P}$-evaluation scheme is said to be *bootstrappable* if it is able to homomorphically evaluate its own decryption function and one additional operation.

The motivation for this is that all FHE schemes we have today are noise-based, where the plaintext is masked by noise. Each multiplication squares that noise, and when the noise passes a certain threshold, decryption fails. However, decryption removes all the noise. Now suppose we have two sets of keys for the same encryption scheme: (sk_1, pk_1, evk_1) and (sk_2, pk_2, evk_2), where evk_2 contains the secret key of the first key set encrypted under the public key of the second set: $evk_2 = \mathsf{Enc}(pk_2, sk_1)$. Let c be a ciphertext that is encrypted under the first keyset, and has accumulated some noise from previous operations. So we have $\mathsf{Dec}(sk_1, c) = m$ for some m. Bootstrapping then essentially means encrypting c under pk_2 (so that it is doubly encrypted), and then homomorphically computing the decryption function for the first key set using the encrypted secret key:

$$c' := \mathsf{Eval}(evk_2, \mathsf{Dec}, \mathsf{Enc}(pk_2, c)) = \mathsf{Eval}(\mathsf{Enc}(pk_2, sk_1), \mathsf{Dec}, \mathsf{Enc}(pk_2, \mathsf{Enc}(pk_1, m))).$$

Intuitively, we doubly encrypt the ciphertext so that we can remove the inner encryption layer with the encrypted secret key. Initially, decryption removes all the noise, but evaluating the decryption function introduces its own noise, which often proves very large. Thus, a scheme that can evaluate its own decryption function and an additional operation (so that the scheme remains useful) has the special distinction of being bootstrappable.
Note that FHE schemes usually rely on the *circular security* assumption: Namely, that security is not impaired by choosing $(sk_1, pk_1, evk_1) = (sk_2, pk_2, evk_2)$, i.e., having only one key and thus having the evaluation key be the secret key encrypted under its own public key ($evk = \mathsf{Enc}(pk, sk)$), which allows endless bootstrappings.

1.2.2 Related Notions

In the context of homomorphic encryption, there are many different related notions that can seem very similar, which is why we explain them in this section.

[7] Definition based on [BV11a], Def. 3.5.

1.2.2.1 *Weaker Variants*

Often, it is not necessary to have *fully* homomorphic encryption schemes – if some information about the function p that will be applied to the encrypted data is known at encryption time, it is enough to chose a $\mathbb{P}$-evaluation scheme where $p \in \mathbb{P}$, and $\mathbb{P}$ does not need to be all functions.

Definition 1.10 (Somewhat Homomorphic Encryption Scheme)**.** A $\mathbb{P}$-evaluation scheme is said to be *somewhat homomorphic* if it is correct for some set of permitted polynomials $\mathbb{P}$.

This definition does not require compactness, so the ciphertexts can grow arbitrarily with each operation. Also, the set of permitted polynomials $\mathbb{P}$ can be any (non-trivial) set of polynomials. Note that in particular, encryption schemes that are homomorphic with regard to one operation, like RSA [RSA78], ElGamal [Gam84], Paillier [Pai99] and others, fall into this category.

Definition 1.11 (Levelled Homomorphic Encryption Scheme[8])**.** A $\mathbb{P}$-evaluation scheme is said to be *levelled homomorphic* if Gen takes an auxiliary input $d \in \alpha$ which specifies the maximum degree of polynomials that can be evaluated. Additionally, the scheme must be correct, compact, and the length of the evaluation output may not depend on d.

This variant is a bit more restricted than somewhat homomorphic encryption, and has a close relation to bootstrapping from Definition 1.9: For bootstrapping, we need to generate a sequence of key sets, where the evaluation key is the secret key of the previous set encrypted under the current public key: $evk_i = \mathsf{Enc}(pk_i, sk_{i-1})$ with evk_0 being empty. Since bootstrapping removes noise that builds up during computation, we need to give an auxiliary parameter d to the algorithm which essentially specifies how many key sets we need, i.e., how often we will need to remove noise.
Note that the above definition did not restrict $\mathbb{P}$ in any way. If we require that the scheme be able to evaluate **all** polynomials up to the specified depth, we obtain the concept of Levelled Fully Homomorphic Encryption:

Definition 1.12 (Levelled Fully Homomorphic Encryption Scheme)**.** A $\mathbb{P}$-evaluation scheme is said to be *levelled fully homomorphic* if it is levelled homomorphic and the set of permitted polynomials $\mathbb{P}$ consists of all polynomials with degree at most d.

1.2.2.2 *Stronger Variants*

Perhaps surprisingly, it turns out that Fully Homomorphic Encryption is not the strongest requirement that we can have. This is due to the fact that while we require the scheme to be able to evaluate any polynomial, the output of Eval does not necessarily need to be a valid input to Eval. In this case, it is not possible to perform further computations once the initial computation has finished – in other words, for any polynomials f and g, one can compute $\mathsf{Eval}(evk, f, c_1, \ldots, c_n)$, $\mathsf{Eval}(evk, g, c_1, \ldots, c_m)$, and $\mathsf{Eval}(evk, g \circ f, c_1, \ldots, c_n)$, but not $\mathsf{Eval}(evk, g, \mathsf{Eval}(evk, f, c_1, \ldots, c_n))$. Thus, the whole function must be computed at once, and results cannot be input into future computations.
Schemes that are able to evaluate outputs of Eval thus have an even stronger characterization, depending on how often one can do this successively.

[8] Definition based on [BV11a], Def. 3.6.

Definition 1.13 (i-hop)**.** Let $i \in \mathbb{N}$. A $\mathbb{P}$-evaluation scheme is said to be *i-hop* if it can correctly evaluate j consecutive calls of Eval for all $j \leq i$.

In this case, the number of calls is bounded by an integer – alternatively, it may be bounded by a polynomial depending on the security parameter λ:

Definition 1.14 (Multi-hop)**.** A $\mathbb{P}$-evaluation scheme is said to be *multi-hop* if i-hop correctness holds for all i that are polynomial in the security parameter λ.

If there is no limit to the number of times that we can consecutively call Eval, we have reached the strongest definition:

Definition 1.15 (∞-hop)**.** A $\mathbb{P}$-evaluation scheme is said to be *∞-hop* if i-hop correctness holds for all i.

Note that in practice, these hop-definitions are not very important: Most FHE schemes achieve some form of circuit privacy (see Definition 1.7), which automatically implies some of these properties. Concretely, from [ABC+15] we know the following two facts[9]:

Theorem 1.1. *A fully homomorphic encryption scheme that is statistically circuit private is multi-hop.*

Theorem 1.2. *A somewhat homomorphic encryption scheme which has perfect circuit privacy is ∞-hop.*

This means that we do not have to worry about this distinction in practice, and we will implicitly assume that our FHE schemes are at least multi-hop from now on.

1.2.2.3 *Other Notions*

There exist four other fields of research that seem related to FHE in that they revolve around computations while keeping something secret, and indeed there is some overlap.

- **Functional Encryption [BSW11]** allows the generation of different secret keys, which reveal the value $f(m)$ for some function f encoded in the secret key, where $c = \mathsf{Enc}(pk, m)$. So formally, $\mathsf{Dec}(sk_f, c) = f(m)$. The difference to FHE is a conceptual one: In FHE, anyone with the evaluation key can perform computations of *their* choosing, but cannot see the result because it is encrypted, and only the data owner has the decryption key. In Functional Encryption, the data owner controls the function to be applied in issuing the secret keys, and the result is received by the computing party in the clear.
- **Obfuscation [BGI+01]** is a largely theoretical notion that tries to formalize the concept of a black box: The user can see the input and output of the function, but learns nothing about the steps in between. This would enable the outsourcing of the function to be applied to the encrypted data without divulging the function to the computing party, but there is an additional connection between obfuscation and FHE: Obfuscation could be used to generate an FHE scheme. Concretely, the black box could contain the decryption and encryption keys, and the function to be

[9] The proof of these theorems can also be found in [ABC+15], where they are denoted as Theorems 3 and 4, respectively.

applied. The ciphertexts would then be decrypted inside the black box, the function computed on the plaintexts, and the result would be encrypted again before leaving the black box. However, it is questionable whether this construction would be any more efficient than direct FHE, and the security implications of hiding the decryption key inside a published program would also have to be carefully examined.

- **Secure Multiparty Computation [Yao82]** (MPC) is an area of research that is often considered as a rival of Fully Homomorphic Encryption, as it also allows computations without revealing the underlying inputs. Concretely, there are several parties who each hold some confidential data, and Secure Multiparty Computation allows them to compute a function on their joint data in such a manner that no party learns anything about the inputs of the other parties. This way of computing a function on sensitive data is usually much more efficient than using Fully Homomorphic Encryption, but MPC requires a large number of interactions between the parties. Fully Homomorphic Encryption, on the other hand, requires no interaction during the computation phase, and the computational load resides entirely with the party performing the computation – thus, in the context of outsourcing computations, it is much more attractive for the client because his computational load is non-existent. In addition, a circuit private FHE scheme (see Definition 1.7) also allows the function that is being computed to remain secret.
- **Differential Privacy [Dwo06]** is a term that quantifies the information that is leaked about an individual by including the data of that individual in a statistical database. On a high level, a small amount of randomness is added to the data in order to create refutability – for example, if a boolean value is being measured, the value will be flipped with a small probability, and an attacker will not know for certain whether the value is the true value or not. The focus here is different than FHE: In Differential Privacy, a function (e.g. the mean or a sum) is computed on a dataset, and the result is manipulated in a way such that individual data cannot be inferred. Of course, changing the result reduces its accuracy, which is why Differential Privacy is used primarily on large datasets where the randomness that is added can be small relative to the result while still ensuring privacy. In contrast, with Fully Homomorphic Encryption the data is encrypted, and then *any* function that can be computed on the plaintext space of the encryption scheme can also be computed on the data, and the encrypted result will be exact.

This concludes the theoretical background on Fully Homomorphic Encryption, and we will now briefly cover existing schemes and implementations before moving on to the motivation for the work in this thesis.

1.2.3 Existing Schemes and Implementations

As already mentioned, the first Fully Homomorphic Encryption Scheme was proposed in [Gen09] in 2009 after being an open problem for over 30 years. Since then, many schemes have followed, greatly improving efficiency from about 30 minutes per operation in [GH11] in the very beginning to under 0.1 seconds in [CGGI16] for the currently fastest scheme.
Conceptually, FHE schemes are often categorized into three different generations. In the following, we briefly present existing cryptosystems for each of the generations (following

the categorization from [BBL17]), along with asymptotic performance, the problem hardness is based on (which we explain in Section 1.2.3.4), and concrete runtimes when known. Values for some early schemes are taken from [ABC+15]. Note that the authors often provide different runtime analyses for their schemes, so the figures may not be comparable. The concrete experiments have been run by the respective authors on widely different hardware, but still give an indication. Blank cells are, to the best of our knowledge, not publicly known.

1.2.3.1 *First Generation*

The first generation of FHE schemes consisted of a handful of very slow schemes, as presented in Table 1.1. Note that [GH11] is a slightly optimized variant of [Gen09].

Scheme	**Underlying Problems**	**Asymptotic Runtime**	**Concrete Runtime**
[Gen09]	BDDP/SVP ([GH11]) & SSSP	$\mathcal{O}(\lambda^{3.5})$ per gate for bootstrapping in [SS10]. Key generation is $\mathcal{O}(\log(n) \cdot n^{1.5})$ where n is the dimension of the lattice in [GH11]	Bootstrapping: From 30 s for small setting, to 30 min for large setting in [GH11].
[vDGHV10]	AGCD & SSSP	Public key size: $\mathcal{O}(\lambda^{10})$, no gate cost given.	-
[SV10]	PCP & SSSP	Key generation is $\mathcal{O}(\log(n) \cdot n^{2.5})$ where n is the dimension of the lattice, according to [GH11].	Key generation took several hours even for small parameters which do not deliver a fully homomorphic scheme, for larger parameters the keys could not be generated.

Table 1.1: First Generation FHE schemes.

1.2.3.2 *Second Generation*

Shortly after these initial schemes, a new set of schemes based on lattice[10] problems emerged, and this lattice structure allowed them to be more efficient than the first generation constructions. Some of these techniques were then also adopted to the setting of [vDGHV10], yielding new schemes in this area as well. An overview of these schemes is given in Table 1.2.

[10] A lattice is a discrete additive subgroup of $\mathbb{R}^n$ that is isomorphic to $\mathbb{Z}_p$ for some prime p. A lattice is given by a basis, and its points are obtained as all linear combinations of the basis using integer coefficients. Cryptographic schemes based on lattice problems usually utilize the fact that certain problems are hard to solve given a "bad" basis of a lattice, but easy to solve given a "good" basis. Thus, a bad basis for a lattice is used as the public key, and the good basis for the same lattice is the secret key.

Scheme	Underlying Problems	Asymptotic Values	Concrete Runtime
[BV11a]	DLWE	Evaluation key size: $\mathcal{O}(\lambda^{2C}\log(\lambda))$ where C is a very large parameter that ensures bootstrappability.	-
[BV11b]	PLWE	Very cheap key generation, unknown for bootstrapping	-
[CMNT11]	AGCD	Reduced public key size from $\mathcal{O}(\lambda^{10})$ in [vDGHV10] to $\mathcal{O}(\lambda^{7})$.	14 min 33 s for bootstrapping, public key size 802 MB.
[Bra12]	Gap-SVP/LWE	Properties depend on ratio of initial noise B to ciphertext modulus q. To evaluate depth d, one needs $q \approx B \cdot p(\lambda)^d$ where $p(\lambda)$ is a polynomial factor.	The RLWE-variant [FV12] achieves 1.4 ms per addition, 59 ms per multiplication, and 89 ms per bootstrapping in [LN14].
[BGV12]	RLWE	Per-gate computation overhead $\tilde{\mathcal{O}}(\lambda \cdot d^3)$ (where d is the depth of the circuit) without bootstrapping, $\tilde{\mathcal{O}}(\lambda^2)$ with bootstrapping.	36 hours for AES encryption (supercomputer, [GHS12]). Updated implementation: AES-128 encryption 2 seconds/block (batched). With bootstrapping 6 seconds/block. In [HS15]: Vectors of 1024 elements from GF(2^{16}) were bootstrapped in 5.5 minutes, single CPU core.
[CNT12]	DAGCD & SSSP	Public key size: $\mathcal{O}(\lambda^5\log(\lambda))$, no gate cost given	Recryption takes about 11 minutes.
[LTV12]	RLWE	Based on the variant of NTRU [HPS98] presented in [SS11]. Needs non-standard security assumption DSPR, which is removed in [BLLN13].	The scale-invariant RLWE-variant [BLLN13] achieves 0.7 ms per addition, 18 ms per multiplication, and 31 ms per bootstrapping in [LN14] (multi-core).
[BGH13]	LWE	Introduces packing for LWE schemes, i.e., computing the same function on several inputs simultaneously. If one ciphertext needs dimension n, then m plaintexts can be encoded in a ciphertext of dimension $n+m$.	-
[CCK$^+$13]	DAGCD	Introduces packing for schemes based on [vDGHV10]. Ciphertext size grows linearly with number of encoded plaintexts.	12 minutes (amortized) per AES ciphertext on a desktop computer.
[CLT14]	ACGD	Secret key linear in depth d of the circuit to be evaluated (levelled FHE). Ciphertext size: $\tilde{\mathcal{O}}(d^2 \cdot \lambda + \lambda^2)$, public key size: $\tilde{\mathcal{O}}(d^4 \cdot \lambda^2 + \lambda^4)$	23 seconds (batched setting) per AES ciphertext on a desktop computer.
[RC14]	SVP & RLWE	Modulus-switching approach, moduli size independent of circuit depth.	Recryption at 275 seconds on 20 cores with 64-bit security.
[SV14]	BDDP & SSSP	Introduced packing for [SV10].	Different parameter settings, fastest recryption 15 seconds for toy setting (6 processors, 47 GB RAM).

Table 1.2: Second Generation FHE schemes.

1.2.3.3 *Third Generation*

While the schemes from the second generation were much more efficient than the first generation asymptotically, the absolute performance in terms of runtime was still astronomical. With [GSW13], new techniques allowing smaller parameter sizes were introduced, and runtimes decreased several orders of magnitude. The state of the art schemes today fall into this category, and the implementation of [CGGI16] will be the library used for our runtime experiments in later chapters. An overview of these third-generation schemes can be found in Table 1.3.

Scheme	**Underlying Problems**	**Asymptotic Values**	**Concrete Runtime**
[GSW13]	LWE	Ciphertext addition and multiplication are simple matrix additions and multiplications. For lattice dimension parameter n, the scheme can evaluate any depth d NAND-circuit with gate overhead $\tilde{\mathcal{O}}((n \cdot d)^{\omega})$ where $\omega < 2.3727$ is the performance of the matrix multiplication algorithm used.	In [KGV16], an optimized version achieves 0.006 seconds per addition and 0.372 seconds per multiplication for their parameter setting on a CPU, and $2 \cdot 10^{-4}$, resp. $3.477 \cdot 10^{-3}$ seconds on a GPU.
[BV14]	DLWE	Bootstrapping needs $p(\lambda)$ matrix (size 5×5) multiplications, where p is some (large) polynomial.	-
[AP14]	Gap-SVP & SIVP	$\tilde{\mathcal{O}}(\lambda)$ for bootstrapping and gate overhead each, resulting in $\tilde{\mathcal{O}}(\lambda^2)$ total gate complexity	-
[DM15]	LWE	Standard LWE parameters, but new bootstrapping algorithm and NAND-computation algorithm. No asymptotic analysis given.	A bootstrapped NAND-gate is evaluated in 0.69 seconds (non-batched).
[CGGI16]	TLWE	Instead of matrix multiplications, the bootstrapping computations are matrix-vector products.	52 ms per bootstrapping. Further improvements in [CGGI17] yield 13 ms per gate.
[BBL17]	DAGCD	Noise grows polynomially (polynomial depends on parameter choice) in λ with each multiplication or NAND, the noise growth is asymmetrical. Multiplying two ciphertexts has complexity $\gamma^2 \log(\gamma)$, where γ is a parameter from the DAGCD assumption depending on λ.	-

Table 1.3: Third Generation FHE schemes.

1.2.3.4 *Underlying Problems*

We now quickly give an informal overview of the (presumed) hard problems underlying the above schemes, more details can be found in the respective papers.

- **SSSP: Sparse Subset Sum Problem**. As explained in Definition 1.9, removing noise from a ciphertext requires the scheme to evaluate its own decryption circuit. This is usually not easily possible, so a trick (called "squashing") is employed: The secret key is written as a sum of other elements, which are hidden in a much larger set of elements. This large set then becomes part of the public key. The new secret key is only an indicator vector that identifies the set entries involved in the sum that reveals the secret. The SSSP assumption is that it is not feasible to extract the secret from the large set without the indicator vector.

- **SVP: Shortest Vector Problem**. This is the problem of finding the shortest vector in a given lattice. It also comes in a decisional variant **Gap-SVP**, which is parametrized by a value γ, and consists of determining whether there either exists a vector shorter than 1, or all vectors in the lattice have length at least γ.

- **SIVP: Shortest Independent Vector Problem**. The goal is to find the shortest independent vectors in a lattice – in other words, to compute a basis (of a sublattice) of only the shortest vectors.

- **BDDP: Bounded Distance Decoding Problem**. This is a version of the Closest Vector Problem (**CVP**), where a vector is given, and the challenge is to find the point on the lattice that is closest to this vector. The BDDP additionally has a guarantee that the vector is actually very close to the lattice point.

- **LWE: Learning with Errors**. The computational variant is to extract a secret $s \in \mathbb{Z}_q^n$ for some n, q from a set of samples of the form $(a_i, < a_i, s > +e_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ with $a_i \in \mathbb{Z}_q^n$ chosen uniformly at random, and e_i coming from some error distribution. The decisional version **DLWE** is to distinguish the above set of samples from the uniform distribution over $\mathbb{Z}_q^n \times \mathbb{Z}_q$. The generalization to the real torus is denoted **TLWE**.

- **RLWE: Ring Learning with Errors**. This is very similar to LWE, but takes place in a ring of the form $R := GF(p)[X]/(f(x)$ for an irreducible polynomial $f(x)$. The samples then have the form $(a_i(x), a_i(x) \cdot s(x) + e_i(x) \in R^2$ for appropriate distributions of e_i and a_i. The computational variant is to extract the secret polynomial $s(x)$, the decisional variant is to distinguish such samples from those that were drawn from the uniform distribution over R^2. The variant **PLWE** is essentially the same, but has some further constraints on the polynomial $f(x)$.

- **AGCD: Approximate Greatest Common Divisor Problem**. In this problem, the task is to find a secret number p given polynomially many near multiples of p in the form $q_i \cdot p + r_i$, with r_i much smaller than $q_i \cdot p$. In the decisional variant **DAGCD**, there is an additional integer $z = x + b \cdot \alpha$, where x is of the same form $q \cdot p + r$, α is from some interval depending on the parameters, and $b \in \{0, 1\}$. The task is now given z and the near multiples of p, to determine whether $b = 0$ or $b = 1$ – in other words, whether z is also a near multiple of p or not.

- **PCP: Polynomial Coset Problem**. This is a decisional problem that asks whether a given value r is the evaluation of a secret polynomial $R(x)$ (subject to some constraints) on a known input α, reduced mod p where p is known, or whether the value r was randomly drawn from $\mathbb{Z}_p$. Formally: Given r, α and p, decide whether $r = R(\alpha) \bmod p$ for some polynomial R, or whether r was drawn randomly from $\mathbb{Z}_p$.

1.2.3.5 *Implementations*

Lastly, we present the existing implementations of Fully Homomorphic Encryption schemes. We limit ourselves to those libraries that are publicly available.

- **Coron's DGHV [LIBa]:** Written in SAGE, this library implements [CNT12], which is an optimization of [vDGHV10].
- **FHEW [LIBb]:** This library implements the scheme from [DM15].
- **FV vs YASHE [LIBc]:** This is the library accompanying the paper [LN14], which implements both [FV12] and [BLLN13]. It is intended as an experimental library, not optimized for public use.
- **HElib [LIBd]:** This was, to our knowledge, the first publicly available FHE library. It implements the scheme from [BGV12].
- **SEAL [LIBe]:** Developed by Microsoft, this library implements an optimized version of [FV12] described in [CLP17]. The library has no external dependencies and is possibly the most user-friendly of the implementations.
- **TFHE [LIBf]:** The currently fastest library, it implements [CGGI16] and the optimizations from [CGGI17]. This is the library we use in later sections.

1.3 FROM THEORY TO PRACTICE

Now that we have seen a formal presentation of FHE and an overview of the existing landscape, we move on to more practical aspects. Colloquially, it is often said that FHE allows the execution of any function on encrypted data, yet performing a division or taking the square root remain open problems in this field. We will see in a second that what seems like a contradiction at first glance is actually due to the imprecision of the colloquial wording.

1.3.1 The Importance of Encoding

Concretely, suppose there is a function $g : S^n \to S$ for some domain S (e.g., $S = \mathbb{R}$) which a user wants to evaluate. However, FHE schemes have finite fields[11] $GF(p^k)$ as their plaintext spaces – some restricted to $GF(2)$, some allowing arbitrary fields. In any case, no FHE scheme (or any encryption scheme, to be exact) has a plaintext space of $\mathbb{N}$ or $\mathbb{Q}$ or $\mathbb{R}$, or more generally S if S can be chosen arbitrarily. Denoting the plaintext space of the Fully Homomorphic Encryption scheme with $\mathcal{P}$ and the ciphertext space[12] with $\mathcal{C}$, we thus first need to map the domain S to a tuple of plaintexts and find a function f that does the same thing as g, but operates on this tuple of plaintexts rather than S. Fully Homomorphic Encryption now promises that there is a function f^* that operates on a tuple of ciphertexts and computes the same thing as f. Note that this is also the source of the perceived contradiction mentioned above: FHE allows a user to compute any function *that they can compute on the plaintext space* on encrypted data instead – if the function cannot be expressed on the plaintext space, Fully Homomorphic Encryption offers no such guarantees. We have illustrated the different conceptual components involved in homomorphically computing a function in Figure 1.1.

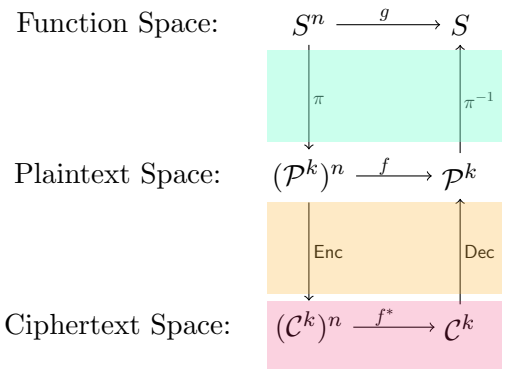

Figure 1.1: Steps in homomorphic evaluation

Fully Homomorphic Encryption tells us that if the upper half of the diagram is commutative, then so is the entire diagram. We see that there are three main components to FHE computations: Mapping the function domain into the plaintext space (upper rectangle, green), encrypting the plaintexts (middle rectangle, orange), and running the function on the ciphertexts (lower rectangle, pink), which often involves some effort in expressing g in

[11] A finite field is an algebraic structure of the form $GF(p^k) = \mathbb{Z}_p[X]/(f(X))$ for a prime p and an irreducible polynomial $f(X)$ of degree k. Its elements are of the form

$$\sum_{i=0}^{k-1} a_i X^i \text{ with } a_i \in \{0, 1, \ldots, p-1\},$$

i.e., all polynomials of degree at most $k-1$ with coefficients in $\{0, 1, \ldots, p-1\}$. Computations are performed using regular polynomial addition and multiplication, reducing by the coefficients by p and the monomials by $f(X)$.

[12] Recall that in Definition 1.1, we differentiated between the space of fresh ciphertexts generated by encryption, $\mathcal{X}$, and the space of evaluation outputs, $\mathcal{Y}$. Since we implicitly assume circuit privacy, these two spaces fall together, so that the notion of a ciphertext space is well-defined.

a way such that it can run on the ciphertexts – i.e., tranforming g into f (resp. f^*).
Research on Fully Homomorphic Encryption schemes and improving their efficiency forms the orange box - this aspect was covered in Section 1.2. Our work focuses on the remaining two boxes: We first analyze the costs of different encoding choices with respect to the underlying plaintext space $GF(p^k)$ for natural numbers, integers and rational numbers. This comprises the green upper box. We then apply our results to concrete algorithms from the field of Machine Learning, which usually run on rational numbers. We show how to move from rationals to finite fields for these cases, and modify the algorithms to allow more efficient evaluation. This constitutes the pink lower box.

1.3.2 Cost Metrics

When we analyze the costs of different encoding bases for FHE computations, the first question that comes to mind is how to define the cost. As it turns out, there is no definitive answer to this question, as there are different possible cost metrics.
Suppose a number is encoded by a set of digits, which are finite field elements. Then encrypting the number means encrypting each digit separately, and computing on the encrypted number is achieved through manipulation of the individual encrypted digits. Thus, we need to emulate the original computation (e.g., adding two numbers) through finite field operations on the plaintext digits that result in a sequence of digits encoding the correct result of the computation. This digit-wise computation can then also be performed on the encrypted digits. Consequently, we express the effort with regard to the underlying finite field operations although in reality, they would be performed on the encrypted digits in the ciphertext space.
In this work, we examine the following three cost metrics:

- **Multiplicative Depth:** As we have mentioned in the context of Definition 1.9, all FHE schemes that we have today are noise-based, in that the plaintext is masked with noise which accumulates with each operation. Concretely, each multiplication of ciphertexts doubles the length of the noise, and each addition increases the length by 1. When the noise passes a certain threshold, decryption fails and the result is lost. The bootstrapping procedure can remove this noise, but it is a very slow procedure and often the bottleneck of the computation. For this reason, minimizing the number of bootstrappings by minimizing the number of consecutive multiplications, also known as multiplicative depth, has frequently been a goal in FHE computations. As an example, consider the product of 4 ciphertexts $c_1 \cdot c_2 \cdot c_3 \cdot c_4$. Then the multiplicative depth depends on the order in which we perform the multiplications: If we compute $c_1 \cdot (c_2 \cdot (c_3 \cdot c_4))$, we get a multiplicative depth of 3. However, if we were to compute the product as $(c_1 \cdot c_2) \cdot (c_3 \cdot c_4)$, we instead get a depth of only 2. Multiplicative depth is closely related to the concept of Levelled FHE (Definition 1.12), where the expected number of bootstrappings determines the number of levels. It can be considered as the most traditional cost metric in this field.

- **Number of Field Multiplications:** For most FHE schemes, ciphertext multiplications are much more expensive than ciphertext additions – so much so that additions are considered "free" by comparison. Thus, it naturally makes sense to keep track of this number. Additionally, when using p-adic encoding, the multiplicative depth quickly becomes so large that bootstrapping is unavoidable, so that minimizing the

total number of multiplications can significantly speed up computations. To see this, consider a parameter setting where there is a bootstrapping operation performed after every multiplication. In this case, the depth is completely irrelevant, and the total number of multiplications completely dominates the computation time. This line of reasoning also extends to less extreme parameter settings, so we regard this as an important metric as well.

- **Number of Field Additions:** For the first and second generation FHE schemes, ciphertext multiplications were a lot more costly than ciphertext additions, but we have always pointed out ([JA16], [JA17]) that there is no theoretical reason why this must be the case. With [CGGI16], there now indeed exists a scheme where every gate is bootstrapped after execution (because they were built from the even more elementary NAND-gate specifically supported by that scheme), and thus there is no significant difference between the two. Thus, we also keep track of the total number of ciphertext additions that need to be performed.

1.3.3 Related Work on FHE Encodings

In the last few years, there has been a large increase in papers about encodings for Fully Homomorphic Encryption. This illustrates the importance of this field, and we give a short overview of these works at this point.

One of the first works to address this topic was [NLV11], where integers are encoded as polynomial ring elements in a bitwise, straightforward fashion. In contrast, our work also accommodates rational numbers instead of being restricted to integers. [CK16] uses continued fractions to encode rational numbers, but is restricted to positive rationals and evaluating linear multivariate polynomials, whereas we allow signed rationals and arbitrary functions. [CSVW16] and [DGBL$^+$15] focus on most efficiently embedding a computation into a single large plaintext space, and the latter also offers an implementation. However, embedding the computation into a large plaintext space is actually *Somewhat* Homomorphic Encryption (see Definition 1.10), as opposed to Fully Homomorphic Encryption as in our work: The plaintext space must be chosen to accommodate the computations, and thus the possible computations are in turn limited by the plaintext space. An extension of [FV12] that allows floating point numbers is presented in [AN16], and [CG15] gives a high-level overview of arithmetic methods for FHE, but resticted to positive numbers rather than arbitrary rationals as in our work. In [GC14], integers are encoded by modifying the underlying scheme [BV11b] to allow a natural mapping from an integer quotient ring, which is of course highly specific to that one encryption scheme, whereas our results apply to an FHE scheme with a finite field plaintexts space (i.e., all currently known schemes). [BBB$^+$17] explores a non-integral base encoding, and [XCWF16] presents different arithmetic algorithms including a costly division, though again limited to positive numbers.

To our knowledge, the only work concerned with the costs of encoding in a base other than $p = 2$ is [KT16], which exclusively analyzes [NK15] and uses different cost metrics than we do. The latter also presents a formula for the carry of a half adder, but merely considers $GF(p^k)$ for $k = 1$ in the context of homomorphically computing the decryption step (needed for bootstrapping) of their variation of [vDGHV10], and does not include an effort analysis. In [CLPX17], an encoding that allows high-precision computations on

rational numbers, but is highly specific to the underlying (new) variant of the [FV12] encryption scheme, is presented. Lastly, [CKKS16] allows approximate operations by utilizing noise from the encryption itself.

1.4 OUTLINE

The remainder of this work is structured as follows: In the following chapters, we will show how to embed natural numbers (**Chapter 2**), integers (**Chapter 3**) and rational numbers (**Chapter 4**) into different FHE plaintext spaces – concretely, we will analyze the costs of different encoding choices with respect to the underlying plaintext space $GF(p^k)$. We thus present optimal choices in p and k for the cost metrics of number of field additions and multiplications, and show that there is no optimum for the metric of multiplicative depth.
In **Chapter 5**, we apply our results to concrete algorithms from the field of Machine Learning. These algorithms usually run on rational numbers, and it is not trivial to turn them into corresponding functions on finite fields. In fact, in some cases (like Section 5.4), it may even be necessary to change the underlying algorithm to allow an efficient mapping into the plaintext space.

Chapter 2

COMPUTING ON NATURAL NUMBERS

In this chapter, we examine the costs for different choices of encoding bases for natural numbers – that is, unsigned integers. Concretely, we want to encode a natural number so that we can then encrypt the digits separately and perform computations on them. The underlying idea is that on the one hand, the involved functions become more complex with a larger encoding base, but on the other hand, fewer digits are required to encode the number, thus reducing complexity. We will see that the first aspect dominates, and the costs increase with larger encoding bases.
To this end, we will first give an introduction to p-adic encoding in **Section 2.1**. We derive the formula to compute the carry digit in the addition of two numbers in **Section 2.2** – addition is an important building block, e.g. for multiplication, and computing the carry is the main challenge in addition. We also calculate the effort of evaluating this formula for the carry on encrypted data in **Section 2.3**. In **Section 2.4**, we use these results to derive the total effort of adding and multiplying natural numbers in the different encoding bases, while also showing how to perform these computations in the optimal way. Lastly, we extend our analysis to more complex finite fields $GF(p^k)$ for $k > 1$ in **Section 2.5**. A summary of the results of this chapter is given in **Section 2.6**.
We will see that $p = 2$ is the best choice for total number of additions and multiplications, and there is no generic optimum regarding depth, as this depends highly on the function being computed and the size of the numbers. Since we have argued in Section 1.3.2 that depth quickly becomes irrelevant when bootstrapping is unavoidable, we thus focus on the otherwise optimal encoding base $p = 2$ in the rest of this work.
This chapter is largely taken from [JA17].

2.1 p-ADIC ENCODING

In day-to-day life, we usually work with numbers in base 10. This choice, however, is arbitrary – computers work in base 2 (binary encoding), hexadecimal representation uses base 16, and in principle any number can be used as a base. When this base is a prime number p, the underlying structure $\mathbb{Z}_p$ is a field, which has the property that every element except 0 is multiplicatively invertible. This is necessary if one wants to perform complex operations on these numbers.
So suppose we fix a prime p as the base. Then we can write any natural number a as a

sum of powers of p, where the coefficients are less than p:

$$a = \sum_{i=0}^{n} a_i \cdot p^i \tag{2.1}$$

with $a_i \in \{0, \ldots, p-1\}$. Then the p-adic representation of a is $(a_n a_{n-1} \ldots a_1 a_0)$.
The following example illustrates this concept:

Example 2.1: *Consider the base $p = 7$ and the decimal number $a = 163$. Then we can write $163 = 147 + 14 + 2 = 3 \cdot 7^2 + 2 \cdot 7^1 + 2 \cdot 7^0$, so the 7-adic representation of 163 is (322).*
Conversely, consider the number (123) in 5-adic encoding. To get back to the more familiar base 10, we write $(123) = 1 \cdot 5^2 + 2 \cdot 5^1 + 3 \cdot 5^0 = 25 + 10 + 3 = 38$. ———

In the rest of this chapter, we will examine the effect of the choice of p on the cost of computating on numbers encoded in this way, and later extend the analysis to more complex finite fields as encoding bases.

2.2 DERIVING THE FORMULA FOR THE CARRY IN ADDITION

In this section and the following Section 2.3, we lay the theoretical foundation for the effort analysis starting from Section 2.4. More concretely, we derive in this section the formulas for the digits of the sum of two numbers in p-adic encoding.

2.2.1 Overview

Suppose we have two natural numbers encoded p-adically: $a = a_n a_{n-1} \ldots a_1 a_0$ and $b = b_n b_{n-1} \ldots b_1 b_0$. If we wish to add these numbers in this encoding, we can write

$$\begin{array}{lccccccc} & & a_n & a_{n-1} & \ldots & a_2 & a_1 & a_0 \\ + & & b_n & b_{n-1} & \ldots & b_2 & b_1 & b_0 \\ \hline = & c_{n+1} & c_n & c_{n-1} & \ldots & c_2 & c_1 & c_0 \end{array} \tag{2.2}$$

To be able to homomorphically evaluate a function on encrypted data, we need to express the result as a polynomial in the inputs - in this case, we need to be able to write

$$c_i = c_i(a_n, b_n, a_{n-1}, b_{n-1}, ..., a_1, b_1, a_0, b_0) \tag{2.3}$$

for any i, where $c_i(\ldots)$ refers to some polynomial. Clearly, it holds that

$$c_i = a_i + b_i + r_i, \tag{2.4}$$

where $r_0 = 0$, and for $i > 0$, r_i is the *carry* from position $i - 1$. Our goal is to express $r_i(a_{i-1}, b_{i-1}, r_{i-1})$ as a polynomial, which will constitute Theorem 2.1. Addition is defined mod p, and we will often write r_i instead of $r_i(a_{i-1}, b_{i-1}, r_{i-1})$ for simplicity.

We first show in **Lemma 1** that $r_i \in \{0, 1\}$, then use that fact to separate the formula for r_i into $r_i = f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1})$ in **Lemma 2**, which has the lowest depth increase out of all possible formulas because r_{i-1} only has degree 1. We proceed

with **Corollary 2.1**, which derives from Lemma 2, showing that the involved functions are symmetric. We present **Lemma 3** and **Lemma 4**, which will be needed in later proofs, and then define an elementary building block in **Definition 2.1**. **Lemma 5** finally concludes the preliminaries to our main theorem. Lastly, **Theorem 2.1** on page 26 presents the main result of this section, namely the closed formula for r_i.

2.2.2 Formula Derivation

We now proceed with the derivation of the formula for the carry $r_i(a_{i-1}, b_{i-1}, r_{i-1})$.

Lemma 1. *The carry r_i is either 0 or 1 for all i.*

Proof: We prove this by induction over the position $i \in \{0, \dots, n+1\}$.
$i = 0$: This is the first position and thus there is no carry from a previous position, i.e.,

$$r_0 = 0.$$

Now for a general position $i > 0$, suppose it holds that $r_{i-1} \in \{0, 1\}$.
Since $a_k \leq p-1$ and $b_k \leq p-1$ for all k, we have (over the natural numbers, not mod p):

$$a_{i-1} + b_{i-1} + r_{i-1} \leq p - 1 + p - 1 + 1 = 2 \cdot p - 1 < 2 \cdot p.$$

Since this last inequality is a real inequality, the result has the form

$$a_{i-1} + b_{i-1} + r_{i-1} = r_i \cdot p + c_i$$

with $r_i \in \{0, 1\}$ and $c_i \in \{0, \dots, p-1\}$. Thus, we need to add $r_i \in \{0, 1\}$ to the next position in the notation of Equation 2.1 – this is exactly the carry.

□

Having established this, we can now express the carry r_i (over $\mathbb{N}$ rather than $\mathbb{Z}_p$) as:

$$r_i = \begin{cases} 0, & a_{i-1} + b_{i-1} + r_{i-1} \leq p - 1 \\ 1, & a_{i-1} + b_{i-1} + r_{i-1} \geq p \end{cases} \tag{2.5}$$

Next, we show how to elegantly express r_i with minimal degree in r_{i-1}:

Lemma 2. *The polynomial for computing $r_i(a_{i-1}, b_{i-1}, r_{i-1})$ can be written as*

$$r_i = f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1}) \tag{2.6}$$

where f_1, f_2 are polynomials in two variables with

$$f_1(a,b) = \begin{cases} 0, & a + b \leq p - 1 \\ 1, & a + b \geq p \end{cases} \quad \textit{and} \quad f_2(a,b) = \begin{cases} 1, & a + b = p - 1 \\ 0, & \textit{else} \end{cases} \tag{2.7}$$

where these sums are again taken over $\mathbb{N}$ rather than $\mathbb{Z}_p$.

Proof: We know that $r_i = f(a_{i-1}, b_{i-1}, r_{i-1})$ is a polynomial in three variables, and since $r_{i-1} \in \{0, 1\}$ by Lemma 1, the power of r_{i-1} must be at most 1 (if we are looking for the most simple form) since $0^x = 0$ and $1^x = 1$ for all $x \geq 1$, so writing e.g. r_{i-1}^5 would always evaluate to the same result as just r_{i-1}. Thus, we can write

$$r_i = f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1})$$

for some polynomials f_1, f_2 by factoring out r_{i-1}.
For the second part of the claim, consider the function $f_1(a_{i-1}, b_{i-1})$:
Since the other half of the equation for r_i is multiplied with r_{i-1}, it evaluates to 0 when $r_{i-1} = 0$. Thus, f_1 defines the behavior of the function when the carry-in r_{i-1} is 0, so it must hold that

$$f_1(a, b) = \begin{cases} 0, & a + b \leq p - 1 \\ 1, & a + b \geq p. \end{cases} \tag{2.8}$$

Now consider the second case where $r_{i-1} = 1$, i.e., $r_i = f_1(a, b) + f_2(a, b)$. This means that we have

$$r_i = \begin{cases} 0, & a + b + 1 \leq p - 1 \\ 1, & a + b + 1 \geq p. \end{cases} = \begin{cases} 0, & a + b \leq p - 2 \\ 1, & a + b \geq p - 1. \end{cases}$$

Comparing this with the above values for $f_1(a, b)$ from Equation 2.8, we see that they are nearly identical and differ solely when $a + b = p - 1$, which results in a carry-out of $r_i = 1$ if the carry-in is $r_{i-1} = 1$. This difference thus constitutes f_2:

$$f_2(a, b) = r_i - f_1(a, b) = \begin{cases} 1, & a + b = p - 1 \\ 0, & \text{else.} \end{cases} \tag{2.9}$$

□

We now continue with a corollary from this Lemma, which we will need in the proof of our main theorem.

Corollary 2.1. *Both $f_1(a, b)$ and $f_2(a, b)$ are symmetric*[1]*. By extension, $r_i(a, b, r_{i-1})$ itself is also symmetric in a and b, i.e., $r_i(a, b, r_{i-1}) = r_i(b, a, r_{i-1})$.*

Proof: The first claim follows directly from the definition of the two functions f_1 and f_2 in Equation 2.7. The second claim can then easily be seen by applying this to Equation 2.6:

$$\begin{aligned} r_i(a, b, r_{i-1}) &= f_1(a, b) + r_{i-1} \cdot f_2(a, b) \\ &= f_1(b, a) + r_{i-1} \cdot f_2(b, a) = r_i(b, a, r_{i-1}). \end{aligned}$$

□

[1] A function $f(a, b)$ is called symmetric if $f(a, b) = f(b, a)$ for all a, b.

Recall the following well-known fact:

Lemma 3. *Over $\mathbb{Z}_p$, it holds that*

$$\prod_{x \in \mathbb{Z}_p^*} x = -1,$$

where $\mathbb{Z}_p^*$ are the invertible elements of $\mathbb{Z}_p$.

Proof: The equation $x^2 = 1$ has exactly two roots over $\mathbb{Z}_p$: 1 and $p-1$. Thus, for every other element $x \in \mathbb{Z}_p^* \backslash \{1, p-1\}$, it holds that $x^{-1} \neq x$. This means that $\prod_{x \in \mathbb{Z}_p^*} x$ contains the inverse of every element except 1 and $p-1$ (i.e., of all $x \in \{2, \ldots, p-2\}$). Thus, all elements except $p-1$ "cancel out" by being multiplied with their inverse (as $\mathbb{Z}_p$ is commutative), so we get $\prod_{x \in \mathbb{Z}_p^*} x = p-1$.
Since $p-1 = -1$ in $\mathbb{Z}_p$, this proves the claim.

□

Note that this fact actually holds over any finite field K, not just those of the form $K = \mathbb{Z}_p$. When $K = \mathbb{Z}_p$, we can also write $\prod_{x \in \mathbb{Z}_p^*} x = (p-1)!$.
The following Lemma will also aid us in the proof of our main theorem:

Lemma 4. *For all $i \in \{0, ..., p-1\}$, it holds that*

$$\prod_{\substack{j=0 \\ j \neq i}}^{p-1} (i-j)^{-1} = p - 1 \bmod p.$$

Proof: We rearrange the product in the following way, with all computations mod p:

$$\prod_{\substack{j=0 \\ j \neq i}}^{p-1} (i-j)^{-1} = \prod_{j=1}^{p-1} j^{-1} = \left(\prod_{j=1}^{p-1} j \right)^{-1} = \Big((p-1)! \Big)^{-1} \overset{*}{=} (p-1)^{-1} = p-1$$

The equality marked with $*$ follows from Lemma 3.

□

To simplify notation, we define the following expression:

Definition 2.1. In the following, denote

$$l_i(x) := \prod_{\substack{j=0 \\ j \neq i}}^{p-1} (x-j).$$

These functions were in fact derived through a bilinear Lagrange-approximation in two variables over the finite field $\mathbb{Z}_p$, which we will see in the proof of Theorem 2.1. With δ_{ik} denoting the Kronecker-delta, which is 1 when $i = k$ and 0 otherwise, we now present the last lemma before this main theorem:

Lemma 5. *For all $i, k \in \{0, \ldots, p-1\}$ it holds that*

$$l_i(k) = -\delta_{ik} = \begin{cases} p-1, & i = k \\ 0, & \textit{otherwise.} \end{cases}$$

Proof: Let $k \neq i$. Then the term $(x - k)$ is a factor in the product, so evaluating at $x = k$ yields a factor of $(k - k) = 0$, thus making the whole product zero.
Now suppose $k = i$. Then much like in the proof of Lemma 4, we have that

$$\prod_{\substack{j=0 \\ j \neq i}}^{p-1} (i - j) = \prod_{j=1}^{p-1} j = (p-1)! = p - 1 \bmod p.$$

□

We now state the formula for the carry bit using these $l_i(x)$-functions:

Theorem 2.1. *The formula for computing $r_i(a_{i-1}, b_{i-1}, r_{i-1})$ is*

$$r_i(a, b, r_{i-1}) = \sum_{k=1}^{p-1} \Big(l_k(b) \cdot \sum_{j=1}^{k} l_{p-j}(a) \Big) + r_{i-1} \cdot (p-1) \cdot l_{p-1}(a+b).$$

This polynomial is unique in that there is no other polynomial of smaller or equal degree which also takes on the correct values for r_i at all points $(a_{i-1}, b_{i-1}, r_{i-1})$ with $a_{i-1}, b_{i-1} \in \{0, \ldots, p-1\}, r_{i-1} \in \{0, 1\}$.

Proof:
Correctness: With the notation of Lemma 2, we only need to show that

$$f_1(a, b) = \sum_{i=1}^{p-1} \Big(l_i(b) \cdot \sum_{j=1}^{i} l_{p-j}(a) \Big) \tag{2.10}$$

and

$$f_2(a, b) = (p-1) \cdot l_{p-1}(a+b). \tag{2.11}$$

We start with f_2: Since

$$f_2(a, b) := \begin{cases} 1, & a + b = p - 1 \\ 0, & \text{else} \end{cases}$$

by Lemma 2, and

$$l_{p-1}(x) := \begin{cases} p-1, & x = p-1 \\ 0, & x \neq p-1, \end{cases}$$

by Lemma 5, it holds that

$$(p-1) \cdot l_{p-1}(x) = \begin{cases} (p-1)^2 = 1, & x = p-1 \\ 0, & \text{else.} \end{cases}$$

Substituting $(a+b)$ for x, we get

$$(p-1) \cdot l_{p-1}(a+b) = \begin{cases} 1, & a+b = p-1 \\ 0, & a+b \neq p-1 \end{cases} \; = \; f_2(a,b).$$

Moving on to $f_1(a,b)$, let $a = \tilde{a}$ and $b = \tilde{b}$ be fixed but arbitrary. We first observe that according to Lemma 5, all $l_i(\tilde{b})$ with $i \neq \tilde{b}$ evaluate to zero.

Since both sums involved in Equation 2.10 start at 1 rather than 0, the sums evaluate to 0 if $\tilde{a} = 0$ (then the terms of the inside sums are all 0) or $\tilde{b} = 0$ (then the coefficients of the outside sum are all 0). This is as it should be, as it can easily be seen that $\tilde{a} + \tilde{b} \leq p-1$ if either $\tilde{a}$ or $\tilde{b}$ is 0, so that $f_1(\tilde{a}, \tilde{b}) = 0$ according to Lemma 2.

Now assume that $\tilde{a}$ and $\tilde{b}$ are both not 0. Then all $l_i(\tilde{b}) = 0$ except $l_{\tilde{b}}(\tilde{b}) = p-1$ (by Lemma 5). We can thus rewrite the right part of Equation 2.10 as

$$\sum_{i=1}^{p-1} \left(l_i(\tilde{b}) \cdot \sum_{j=1}^{i} l_{p-j}(\tilde{a}) \right) = l_{\tilde{b}}(\tilde{b}) \cdot \sum_{j=1}^{\tilde{b}} l_{p-j}(\tilde{a}) = (p-1) \cdot \sum_{j=1}^{\tilde{b}} l_{p-j}(\tilde{a}). \tag{2.12}$$

Now we distinguish two cases:
Case 1: $\tilde{a} + \tilde{b} \leq p-1$.
This means that $f_1(\tilde{a}, \tilde{b}) = 0$ according to Lemma 2. We also have

$$\tilde{a} + \tilde{b} \leq p-1 \Leftrightarrow \tilde{a} \leq p - \tilde{b} - 1. \tag{2.13}$$

Looking at the inner sum of Equation 2.10, we can write it out as

$$l_{p-1}(\tilde{a}) + l_{p-2}(\tilde{a}) + \cdots + l_{p-\tilde{b}}(\tilde{a}).$$

Since $\tilde{a} < p - \tilde{b}$ (Equation 2.13), $l_{\tilde{a}}(\tilde{a})$ is not included in this sum, and thus all terms of the inner sum evaluate to 0 (Lemma 5), making the whole sum 0 when $\tilde{a} + \tilde{b} \leq p-1$:

$$\sum_{i=1}^{p-1} \left(l_i(\tilde{b}) \cdot \sum_{j=1}^{i} l_{p-j}(\tilde{a}) \right) = l_{\tilde{b}}(\tilde{b}) \cdot \sum_{j=1}^{\tilde{b}} l_{p-j}(\tilde{a}) = (p-1) \cdot \sum_{j=1}^{\tilde{b}} 0 = (p-1) \cdot 0 = 0. \tag{2.14}$$

Case 2: $\tilde{a} + \tilde{b} \geq p$.
This means that $f_1(\tilde{a}, \tilde{b}) = 1$ according to Lemma 2.

By the same argument as in Case 1, we have

$$\tilde{a} + \tilde{b} \geq p \Leftrightarrow \tilde{a} \geq p - \tilde{b}. \tag{2.15}$$

Thus, $l_{\tilde{a}}(\tilde{a})$ is included in the written-out inner sum

$$l_{p-1}(\tilde{a}) + l_{p-2}(\tilde{a}) + \cdots + l_{p-\tilde{b}}(\tilde{a}),$$

which evaluates to $p-1$ according to Lemma 5. Combining this with Equation 2.12, the sum now evaluates to

$$\begin{aligned}
\sum_{i=1}^{p-1} \Big(l_i(\tilde{b}) \cdot \sum_{j=1}^{i} l_{p-j}(\tilde{a}) \Big) &= l_{\tilde{b}}(\tilde{b}) \cdot \sum_{j=1}^{\tilde{b}} l_{p-j}(\tilde{a}) \\
&= (p-1) \cdot \sum_{j=1}^{\tilde{b}} l_{p-j}(\tilde{a}) \\
&= (p-1) \cdot l_{\tilde{a}}(\tilde{a}) \\
&= (p-1) \cdot (p-1) \\
&= 1,
\end{aligned} \tag{2.16}$$

where computations are mod p.
Combining equations 2.14 and 2.16, we get:

$$\sum_{i=1}^{p-1} \Big(l_i(\tilde{b}) \cdot \sum_{j=1}^{i} l_{p-j}(\tilde{a}) \Big) = \begin{cases} 0, & a+b \leq p-1 \quad \text{(Case 1)} \\ 1, & a+b \geq p \quad \text{(Case 2)} \end{cases} = f_1(a,b). \tag{2.17}$$

Uniqueness: To prove uniqueness, we take quick look at how our polynomial f_1 was derived by recalling Lagrange's polynomial interpolation: Given $k+1$ points

$$\big(x_i, y_i = f(x_i)\big), i = 0, \ldots, k,$$

the Lagrangian interpolation polynomial (which interpolates the function f through these points) is given as

$$L(x) = \sum_{i=0}^{k} h_i(x) \cdot y_i, \text{ where } h_i(x) := \prod_{\substack{j=0 \\ j \neq i}}^{k} \frac{x - x_j}{x_i - x_j}.$$

The idea now is to perform a bivariate Lagrangian interpolation of $f_1(a,b)$ by first performing p interpolations over $\mathbb{Z}_p$ to obtain the functions $f_b = f_1|_b(\cdot) = f_1(\cdot, b)$:

$$f_b(a) = \sum_{i=0}^{p-1} h_i(a) \cdot f(i,b).$$

Then, using these polynomials as "values" for f_1 at the points $b = 0, \ldots, p-1$, we perform a second interpolation over $\mathbb{Z}_p[a]$ to obtain $f_1(a,b)$:

$$f_1(a,b) = \sum_{i=0}^{p-1} h_i(b) \cdot f_i(a) = \sum_{i=0}^{p-1} h_i(b) \cdot \Big(\sum_{j=0}^{p-1} h_j(a) \cdot f(j,i) \Big). \tag{2.18}$$

Now note the following: Since we are given the values of the function on all values in $\mathbb{Z}_p$ and are also computing in this field, we can write

$$h_i(x) := \prod_{\substack{j=0 \\ j\neq i}}^{k} \frac{x - x_j}{x_i - x_j} = \prod_{\substack{j=0 \\ j\neq i}}^{p-1} \frac{x - j}{i - j}.$$

Using Lemma 4, we see that

$$h_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{p-1} \frac{x - j}{i - j} = (p-1) \cdot \prod_{\substack{j=0 \\ j\neq i}}^{p-1} (x - j) = (p-1) \cdot l_i(x).$$

Now we can rewrite Equation 2.18 as

$$\begin{aligned} f_1(a,b) &= \sum_{i=0}^{p-1} (p-1) \cdot l_i(b) \cdot \Big(\sum_{j=0}^{p-1} (p-1) \cdot l_j(a) \cdot f(j,i) \Big) \\ &= (p-1)^2 \cdot \sum_{i=0}^{p-1} l_i(b) \cdot \sum_{j=0}^{p-1} l_j(a) \cdot f(j,i) \\ &= \sum_{i=0}^{p-1} l_i(b) \cdot \sum_{j=0}^{p-1} l_j(a) \cdot f(j,i). \end{aligned} \tag{2.19}$$

Lastly, since

$$f(j,i) = \begin{cases} 1, & i + j \geq p \\ 0, & \text{else,} \end{cases}$$

we see that only $l_j(a)$ with

$$i + j \geq p \Leftrightarrow j \geq p - i \Leftrightarrow j \in \{p - i, \ldots, p - 1\}$$

are multiplied with $f(j,i) = 1$, whereas all other values are multiplied with 0.
Likewise, $f(j,0) = f(0,i) = 0$ for all i, j, so both the outer and inner sums can disregard $i, j = 0$. Thus, we get the formula

$$f_1(a,b) = \sum_{i=1}^{p-1} \Big(l_i(b) \cdot \sum_{j=p-i}^{p-1} l_j(a) \cdot 1 \Big) = \sum_{i=1}^{p-1} \Big(l_i(b) \cdot \sum_{j=1}^{i} l_{p-j}(a) \Big),$$

which is exactly the formula we already proved correctness for.
Looking at this derivation of the formula, we consider the following well-known facts:

1. $\mathbb{Z}_p[a]$ is a factorial ring, as it is a polynomial ring over a field.
2. $(\mathbb{Z}_p[a])[b] \cong \mathbb{Z}_p[a,b]$ is a factorial ring, as it is a polynomial ring over a factorial ring.
3. In a factorial ring (or more generally, an integral domain), a polynomial of degree $n \geq 1$ has at most n roots.

4. If $f(x)$ and $g(x)$ are polynomials of degree at most $p-1$, then $h(x) := f(x) - g(x)$ also has degree at most $p-1$.

Putting all this together, we prove uniqueness in two steps: First, we show that the $f_b(a)$ are unique, then we show that $f_1(a,b)$ is unique.
Let $b \in \{0,\dots,p-1\}$ be fixed but arbitrary and consider the polynomial $f_b(a)$, which was derived through Lagrangian interpolation in the points (a,b) for all $a \in \{0,\dots,p-1\}$ as described above.
Now assume that there is a different polynomial $g(a) \neq f_b(a)$ of equal or lower degree (which is $p-1$) with $g(a) = f_b(a) = f_1(a,b)$ for all $a \in \{0,\dots,p-1\}$.
Then $h(a) := g(a) - f_b(a)$ is a polynomial of degree at most $p-1$ (fact 4) with at least p roots (in $a = 0,\dots,p-1$).
This, however, is a contradiction to $\mathbb{Z}_p[a]$ being a factorial ring (fact 1), since this polynomial can have at most $p-1$ roots in such a ring (fact 3).
Thus, the polynomial $g(a)$ cannot exist.

By exactly the same reasoning (seeing the polynomials $f_b(a)$ as points in $\mathbb{Z}_p[a]$ to perform Lagrangian interpolations, and using fact 2), it can be seen that $f_1(a,b)$ is also unique.

Since $f_1(a,b)$ is symmetric by Corollary 2.1, the bilinear interpolation is well-defined in that it yields the same result when interpolating first over $\mathbb{Z}_p[b]$ in the variable b (yielding polynomials $f_a(b)$) and then over $\mathbb{Z}_p[b]$ to obtain $f_1(a,b)$.
Thus, the polynomial $f_1(a,b)$ is unique.

Regarding the polynomial $f_2(a,b) = (p-1) \cdot l_{p-1}(a+b)$, we write f_2 as a polynomial in one variable $u := a + b$.
Then $(p-1) \cdot l_{p-1}(u)$ has degree $p-1$ and is fixed on p points: It is 0 for $u = 0,\dots,p-2$, and is 1 in $u = p-1$.
Since $\mathbb{Z}_p[u]$ is a factorial ring (fact 1), we can again apply the same reasoning as above: If there were another polynomial of equal or less degree that is also defined in these points, subtracting them would yield a polynomial of degree at most $p-1$ with p roots (fact 4), which cannot be the case in a factorial ring (fact 3).

Thus, $f_2(a,b)$ is also unique, and as such the entire polynomial $r_i(a_{i-1}, b_{i-1}, r_{i-1})$.

□

In the next section, we will use this formula to derive the effort of computing a single digit in adding two natural numbers.

2.3 EFFORT OF EVALUATING THE CARRY FORMULA

We now compute the costs to compute each digit c_i when adding two natural numbers encoded p-adically by analyzing the formula from Theorem 2.1. Recall that

$$c_i = a_i + b_i + r_i.$$

Moreover, it holds that r_i can be computed as

$$r_i = f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1})$$

(see Theorem 2.1). Note that there cannot be any cancellation between the terms of f_1 and f_2 due to the variable r_{i-1}. Putting this together, we get as effort for computing c_i for $i > 1$:

- Field additions: $\text{Adds}(c_i) = 3 + \text{Adds}(f_1) + \text{Adds}(f_2)$
- Field multiplications: $\text{Mults}(c_i) = 1 + \text{Mults}(f_1) + \text{Mults}(f_2)$
- Multiplicative depth: $d(c_i) = \max\{d(f_1), \max\{d(r_{i-1}), d(f_2)\} + 1\}$

Note that $d(f_i)$ denotes the multiplicative depth of f_i, and $\text{Adds}(f_i)$ denotes the number of field additions incurred through the function f_i – i.e., the number of additions we need to perform on the input digits in the finite field that is our encoding base while computing the function f_i. Likewise, $\text{Mults}(f_i)$ is the number of field multiplications. We now need to compute the individual components. The computation is split into three parts:

1. First, the effort for evaluating f_1 is derived in **Subsection 2.3.1**.
2. Then, the effort for f_2 is computed in **Subsection 2.3.2**.
3. Lastly, the results are combined for the total effort for each digit in **Subsection 2.3.3**.

The effort for f_1 is again split into several parts: First, the effort for the closed formula from Theorem 2.1 is examined, and a time-memory tradeoff that minimizes effort by precomputing certain values is presented. Then, we compare this to the expanded form of the formula and include its analysis. This expanded formula suggests a lower optimal depth, so we take the addition/multiplication costs from the closed formula and the lower depth value from the expanded formula as best-case costs to be as unbiased as possible.

2.3.1 Effort for f_1

Recall the formula for f_1:

$$f_1(a, b) = \sum_{i=1}^{p-1} \left(l_i(b) \cdot \sum_{j=1}^{i} l_{p-j}(a) \right).$$

We first present the straightforward way of computation, and then show how to reduce the number of field multiplications in a time-memory tradeoff, and finally compare it to using the expanded form of the polynomial.

2.3.1.1 *Straightforward Approach*

The straightforward computation consists of the following steps:

1. Compute $(a - j)$ and $(b - j)$ for $j = 1, \ldots, p - 1$:
   - Field additions: $2p - 2$
   - Field multiplications: 0
   - Multiplicative depth: $+0$
2. Compute $l_i(a), l_i(b)$ for $i = 1, \ldots, p - 1$ using the precomputed factors from the previous step:
   - Field additions: 0
   - Field multiplications: $2 \cdot (p - 1) \cdot (p - 2) = 2p^2 - 6p + 4$
   - Multiplicative depth: $\lceil \log_2(p - 1) \rceil$ for each $l_i(a)$ and $l_i(b)$
3. Compute $\sum_{j=1}^{i} l_{p-j}(a)$ for $i = 1, \ldots, p - 1$ recursively by setting for $i = 1$:

   $$\sum_{j=1}^{1} l_{p-j}(a) = l_{p-1}(a)$$

   and then computing for $i = 2, \ldots, p - 1$:

   $$\sum_{j=1}^{i} l_{p-j}(a) = \Big(\sum_{j=1}^{i-1} l_{p-j}(a) \Big) + l_{p-i}(a),$$

   which incurs only one field addition for each sum. By computing this way, we get:
   - Field additions: $p - 2$
   - Field multiplications: 0
   - Multiplicative depth: $+0$
4. Compute $l_i(b) \cdot \sum_{j=1}^{i} l_{p-j}(a)$ for $i = 1, \ldots, p - 1$.
   This incurs one multiplication for each i and raises the multiplicative depth by one (recall that when multiplying two factors, each of depth d, the product has depth $d + 1$):
   - Field additions: 0
   - Field multiplications: $p - 1$
   - Multiplicative depth: $+1$, so $\lceil \log_2(p - 1) \rceil + 1$ total
5. Lastly, sum up all the $l_i(b) \cdot \sum_{j=1}^{i} l_{p-j}(a)$ to obtain $\sum_{i=1}^{p-1} \Big(l_i(b) \cdot \sum_{j=1}^{i} l_{p-j}(a) \Big)$:
   - Field additions: $p - 2$
   - Field multiplications: 0
   - Multiplicative depth: $+0$

In total, we obtain the effort as:

Step	**Additions**	**Multiplications**	**Depth**
1	$2p-2$	0	0
2	0	$2p^2-6p+4$	$\lceil \log_2(p-1) \rceil$
3	$p-2$	0	$+0$
4	0	$p-1$	$+1$
5	$p-2$	0	$+0$
Total	$4p-6$	$2p^2-5p+3$	$\lceil \log_2(p-1) \rceil + 1$

2.3.1.2 *Time-Memory Tradeoff*

Step 2 of the straightforward approach is obviously far from efficient, since we are computing all $l_i(a), l_i(b)$ independently from each other in spite of their close relations. As an example, consider

$$l_1(a) = \prod_{\substack{j=0 \\ j \neq 1}}^{p-1} (a-j) = a \cdot (a-2) \cdot (a-3) \cdot \cdots \cdot (a-(p-1))$$

and

$$l_2(a) = \prod_{\substack{j=0 \\ j \neq 2}}^{p-1} (a-j) = a \cdot (a-1) \cdot (a-3) \cdot \cdots \cdot (a-(p-1)).$$

With Step 2 as it is, computing both products would cost us $2 \cdot (p-2)$ multiplications. Imagine, however, if we were to precompute the value

$$L = a \cdot (a-3) \cdot \cdots \cdot (a-(p-1)) :$$

The precomputation can be done with $p-3$ multiplications (because there are $p-2$ factors), and from this precomputed value, $l_1(a) = L \cdot (p-2)$ and $l_2(a) = L \cdot (p-1)$ can be computed with one multiplication each, yielding a total of $p-1$ multiplications instead of $2p-4$.

From the definition of the $l_i(x)$, it is obvious that each $l_i(x)$ contains all factors from $\{x, (x-1), \ldots, (x-(p-1))\}$ except for one (namely $(x-i)$). Now in a first step, suppose we divide the factors into two sets

$$\left\{x, (x-1), \ldots, (x-\left\lfloor \frac{p}{2} \right\rfloor)\right\} \quad \text{and} \quad \left\{(x-\left\lceil \frac{p}{2} \right\rceil), \ldots, (x-(p-1))\right\}$$

and multiply the elements in each set to obtain two intermediate products[2]

$$L_2^1 := \prod_{j=0}^{\lfloor \frac{p}{2} \rfloor} (x-j) \quad \text{and} \quad L_2^2 := \prod_{j=\lceil \frac{p}{2} \rceil}^{p-1} (x-j).$$

Then for each $i = 1, \ldots, p-1$, either L_2^1 or L_2^2 is contained in $l_i(x)$, and conversely each $l_i(x)$ can be calculated from one of the L_2^j with roughly $\frac{p}{2} - 1$ multiplications (because there the are $\frac{p}{2} - 1$ missing factors and L_2^j that need to be multiplied.). Note that of course $\frac{p}{2} - 1$ will not generally be an integer because p is odd (except when $p = 2$), but when adding the efforts, we arrive at the correct result since for an odd number, it holds that $\lfloor \frac{p}{2} \rfloor + \lceil \frac{p}{2} \rceil = p = \frac{p}{2} + \frac{p}{2}$. Thus, for readability, we simply write $\frac{p}{2}$.
Adding the $2 \cdot (\frac{p}{2} - 1)$ multiplications for computing the two L_2^j, we get $(p-1) \cdot (\frac{p}{2} - 1) + 2 \cdot (\frac{p}{2} - 1)$ multiplications for computing all $l_i(a)$. Since we need to do this for $l_i(b)$ as well, we multiply this number by 2 to obtain

$$2 \cdot ((p-1) \cdot (\frac{p}{2} - 1) + 2 \cdot (\frac{p}{2} - 1)) = (p-1) \cdot (p-2) + 2 \cdot (p-2) = p^2 - p - 2$$

field multiplications for Step 2 instead of $2p^2 - 6p + 4$ from before.
Of course, we do not need to stop here: We can also calculate

$$L_4^1 := \prod_{j=0}^{\lfloor \frac{p}{4} \rfloor} (x-j), \ \ldots, \ L_4^4 := \prod_{j=3 \cdot \lceil \frac{p}{4} \rceil}^{p-1} (x-j).$$

Each of these L_4^i can be computed with $\frac{p}{4} - 1$ multiplications (though some of the products L_4^i will contain one factor less, so this is really an upper bound), so we have a total of $4 \cdot (\frac{p}{4} - 1) = p - 4$ multiplications from these intermediate products. Also, we can compute $L_2^1 = L_4^2 \cdot L_4^2$ and $L_2^2 = L_4^3 \cdot L_4^4$ with only 2 further multiplications, so precomputation incurs $p - 2$ multiplications in total.
Now for each of the $l_i(x)$, we can compute

$$l_i(x) = L_2^{j_1} \cdot L_4^{j_2} \cdot \tilde{r} \quad \text{for some} \quad j_1 \in \{1, 2\}, \ j_2 \in \{1, 2, 3, 4\},$$

where $\tilde{r}$ consists of the $\frac{p}{4} - 1$ remaining terms that are multiplied in trivial fashion. Thus, we get a total of $1 + 1 + \frac{p}{4} - 2 = \frac{p}{4}$ multiplications for each l_i from this part of the computation. Putting this together, the multiplication cost of computing $l_i(x)$ for all $i = 1, \ldots, p-1$ in this manner is $p - 2 + (p-1) \cdot \frac{p}{4}$. Since we need to do all this for both variables values a and b, we get a total of

$$2 \cdot (p - 2 + (p-1) \cdot \frac{p}{4}) = \frac{1}{2}p^2 + \frac{3}{2}p - 4$$

[2] Generally, the notation L_k^i denotes that we have divided the p factors into k roughly equal sets, and this is the i^{th} of these sets:

$$L_k^i := \prod_{j=(i-1) \cdot \lceil \frac{p}{k} \rceil}^{i \cdot \lfloor \frac{p}{k} \rfloor} (x-j)$$

field multiplications.
Generalizing this idea, if we split the factors into $k = 2^w$ groups for some w, we observe that each L_k^i has (roughly) $\frac{p}{k}$ elements and can thus be computed with $\frac{p}{k}-1$ field multiplications. Doing this for all k L_k^i's, we get

$$k \cdot (\frac{p}{k} - 1) = p - k$$

multiplications. Also, computing all $L_{k/2}^j$ from the L_k^i only costs additional $\frac{k}{2}$ multiplications. Thus, computing all L_n^j for $n = \frac{k}{2}$ down to $n = 2$ incurs

$$\frac{k}{2} + \frac{k}{4} + \cdots + 2 = \sum_{z=1}^{w-1} 2^z = (\sum_{z=0}^{w-1} 2^z) - 1 = (2^w - 1) - 1 = k - 2$$

field multiplications. In total, precomputation always needs $(p - k) + (k - 2) = p - 2$ multiplications.
In the same way as above, we can now compute

$$l_i(x) = L_2^{j_1} \cdot \cdots \cdot L_k^{j_w} \cdot \tilde{r}$$

for some $j_i \in \{1, \ldots, 2^i\}$ and $\tilde{r}$ consisting of the remaining $\frac{p}{k} - 1$ terms that are multiplied in trivial fashion (incurring $\frac{p}{k} - 2$ multiplications). In addition to the multiplications from $\tilde{r}$, there are further $w = \log_2(k)$ multiplications in the formula for $l_i(x)$, so for each $l_i(x)$ we get $\frac{p}{k} - 2 + \log_2(k)$ field multiplications.
Putting this together and again multiplying by 2 because we need to compute the formulas for the values a and b, we obtain

$$2 \cdot \Big(p - 2 + (p-1) \cdot (\frac{p}{k} - 2 + \log_2(k))\Big) = \frac{2}{k}p^2 + \Big(2\log_2(k) - \frac{2k+2}{k}\Big) \cdot p - 2 \cdot \log_2(k)$$

field multiplications instead of Step 2 in our original effort analysis.
Taking this to the extreme where $k = p/2$, i.e., we precompute everything down to products of 2 factors, we get

$$\begin{aligned}
&\frac{2}{p/2}p^2 + (2\log_2(p/2) - \frac{2(p/2)+2}{p/2}) \cdot p - 2 \cdot \log_2(p/2) \\
&= 4p + (2 \cdot (\log_2(p) - 1) - 2 - \frac{4}{p}) \cdot p - 2 \cdot (\log_2(p) - 1) \\
&= 4p + 2p \cdot \log_2(p) - 4p - 4 + 2 - 2 \cdot \log_2(p) \\
&= 2p \cdot \log_2(p) - 2 \cdot \log_2(p) - 2
\end{aligned}$$

This minimum number of multiplications requires storing

$$2 \cdot (k + \frac{k}{2} + \frac{k}{4} + \cdots + 2) = 2 \cdot (p/2 + \frac{p/2}{2} + \frac{p/2}{4} + \cdots + 2)$$

precomputed values. Setting for simplicity reasons $p \approx 2^w$ for some w, we need to store

$$2 \cdot (2^{w-1} + 2^{w-2} + 2^{w-3} + \cdots + 2) = 2 \cdot ((\sum_{i=0}^{w-1} 2^i) - 1) = 2 \cdot (2^w - 1) - 1 \approx 2p - 4$$

numbers.

Replacing the costs of Step 2 with this new value, we get as effort for computing f_1 with the closed formula:

Step	**Additions**	**Multiplications**	**Depth**
1	$2p-2$	0	0
2	0	$2p \cdot \log_2(p) - 2 \cdot \log_2(p) - 2$	$\lceil \log_2(p-1) \rceil$
3	$p-2$	0	$+0$
4	0	$p-1$	$+1$
5	$p-2$	0	$+0$
Total	$4p-6$	$2p \cdot \log_2(p) + p - 2 \cdot \log_2(p) - 3$	$\lceil \log_2(p-1) \rceil + 1$

Note that the idea of precomputation only really makes sense if $p > 4$, so for $p \in \{2, 3\}$, we use the non-precomputation formulas from the beginning.

2.3.1.3 *Using the Expanded Formula*

We note at this point that it seems as if the polynomial for

$$f_1 = \sum_{i=1}^{p-1} \left(l_i(b) \cdot \sum_{j=1}^{i} l_{p-j}(a) \right)$$

has a smaller degree than the expected $2p-2$ when the double sum is expanded. We will prove some bounds on this degree and discuss the impact of using this expanded formula instead of the closed one. The results of this analysis are that the closed double-sum formula is much more efficient regarding the metrics of field additions and multiplications, and that the best possible depth of $\lceil \log_2(p) \rceil$ can be achieved through the expanded formula at much higher addition and multiplication costs. The difference in multiplicative depth to the closed formula is at most 1.
We begin with a Lemma regarding the form of the l_i-functions when expanded:

Lemma 6. *In the expanded form of $l_i(x)$, the term x^k has the coefficient $i^{p-(k+1)} \bmod p$ where $1 \leq k \leq p-1$ and $i \in \{1, \ldots, p-1\}$. In other words,*

$$l_i(x) = x^{p-1} + i \cdot x^{p-2} + i^2 \cdot x^{p-3} + \cdots + i^{p-3} \cdot x^2 + i^{p-2} \cdot x \bmod p.$$

Proof: It is a well-known fact that over $\mathbb{Z}_p$, it holds that

$$F(x) := \prod_{j=0}^{p-1} (x - j) = x^p - x.$$

Now note that

$$l_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{p-1} (x - j) = \frac{\prod_{i=0}^{p-1} (x - j)}{x - i} = \frac{F(x)}{x - i} = \frac{x^p - x}{x - i}.$$

Performing this division by hand, we get

$$\begin{array}{llllllll}
 & (x^p- & x)/(x-i) = & x^{p-1}+ & i\cdot x^{p-2}+ & \cdots + & i^{p-3}\cdot x^2+ & i^{p-2}\cdot x \\
- & (x^p & -i\cdot x^{p-1}) & & & & & \\
\hline
 & & i\cdot x^{p-1} & & & & & -x \\
 & & -(i\cdot x^{p-1} & -i^2\cdot x^{p-2}) & & & & \\
\hline
 & & & i^2\cdot x^{p-2} & & & & -x \\
 & & & & \cdots & & & \\
\hline
 & & & & & & i^{p-2}\cdot x^2 & -x \\
 & & & & & & -(i^{p-2}\cdot x^2 & -i^{p-1}\cdot x) \\
\hline
 & & & & & & & 0
\end{array}$$

where the last line of 0 occurs because $i \neq 0$ and thus $i^{p-1} = 1 \bmod p$ according to Fermat's Little Theorem.

□

Corollary 6.1. *The coefficient of the term $a^x b^y$ in the expanded polynomial for f_1 is*

$$(p-1)^{-x} \cdot \sum_{i=1}^{p-1} \Big(i^{-y} \cdot \sum_{j=1}^{i} j^{-x} \Big).$$

Proof: We first recall Formula 2.10:

$$f_1(a,b) = \sum_{i=1}^{p-1} \Big(l_i(b) \cdot \sum_{j=1}^{i} l_{p-j}(a) \Big).$$

Using Lemma 6, we can write this as

$$f_1(a,b) = \sum_{i=1}^{p-1} \Big((b^{p-1} + i\cdot b^{p-2} + \cdots + i^{p-2}\cdot b) \cdot \sum_{j=1}^{i} (a^{p-1} + (p-j)\cdot a^{p-2} + \cdots + (p-j)^{p-2}\cdot a) \Big).$$

As can easily be seen from this notation, the term $a^x \cdot b^y$ (with $x, y \in \{1, \ldots, p-1\}$) will have the coefficient

$$\sum_{i=1}^{p-1} \Big(i^{p-y-1} \cdot \sum_{j=1}^{i} (p-j)^{p-x-1} \Big).$$

Since the order in the exponent mod p is $p-1$, we can equivalently write

$$\sum_{i=1}^{p-1} \Big(i^{-y} \cdot \sum_{j=1}^{i} (p-j)^{-x} \Big).$$

Finally, rewriting $(p-j)^{-x} = (p-1)^{-x} \cdot j^{-x}$ and moving the constant value $(p-1)^{-x}$ in front of the sums, we obtain as the coefficient of the term $a^x \cdot b^y$:

$$(p-1)^{-x} \cdot \sum_{i=1}^{p-1} \Big(i^{-y} \cdot \sum_{j=1}^{i} j^{-x} \Big). \tag{2.20}$$

□

We now use this formula to establish bounds on the degree of the expanded form of f_1:

Theorem 2.2. *In its expanded form, f_1 has a degree less than the expected $2p-2$, but at least p.*

Proof: We first show that the coefficient of the term $a^{p-1} \cdot b$ is always $p-1$, so the degree of the polynomial is at least p:
From Corollary 6.1, we know that the coefficient of the term $a^{p-1} \cdot b$ is

$$(p-1)^{-(p-1)} \cdot \sum_{i=1}^{p-1} \Big(i^{-1} \cdot \sum_{j=1}^{i} j^{-(p-1)} \Big).$$

Since both sums start at 1, all involved elements are in $\mathbb{Z}_p^*$ and thus $j^{-(p-1)} = 1 \bmod p$, meaning we can write the inner sum as

$$\sum_{j=1}^{i} j^{-(p-1)} = \sum_{j=1}^{i} 1 = i.$$

Substituting this into the entire formula and noting $(p-1)^{-(p-1)} = 1$ in the front, we get

$$\sum_{i=1}^{p-1} \Big(i^{-1} \cdot i \Big) = \sum_{i=1}^{p-1} 1 = (p-1) \neq 0.$$

Thus, the term $a^{p-1} \cdot b$, which has degree p, has a non-zero coefficient, implying a total degree of at least p.
For the second part of our claim, we now show that the coefficient of the term $a^{p-1} \cdot b^{p-1}$, which is the only term of degree $2p-2$, is always 0.
Using again Corollary 6.1, we know that the coefficient of the term $a^{p-1} \cdot b^{p-1}$ is

$$(p-1)^{-(p-1)} \cdot \sum_{i=1}^{p-1} \Big(i^{-(p-1)} \cdot \sum_{j=1}^{i} j^{-(p-1)} \Big).$$

Again noting that all involved elements g are invertible and thus $g^{p-1} = 1 = g^{-(p-1)}$, we rewrite the coefficient as

$$1 \cdot \sum_{i=1}^{p-1} \Big(1 \cdot \sum_{j=1}^{i} 1 \Big) = \sum_{i=1}^{p-1} i = \frac{(p-1)}{2} \cdot p = 0 \bmod p$$

using the Gaussian sum formula[3].

□

Experimental results suggest the following conjecture:

Conjecture 1. *f_1 has exactly degree p.*

The conjecture held for all p that we tested (all primes under 500, and some additional larger ones), and we will work with this value when comparing the performance of the expanded form to that of the closed form. This is justified because the conjecture value is equal to our lower bound on the degree from Theorem 2.2. As we will see in the following, the closed formula will turn out to be the better choice even when assuming this lower bound.
Nonetheless, we present this expanded form of f_1 for the first few primes:

$$\begin{aligned}
p = 2 : f_1(a,b) = &\quad ab \\
p = 3 : f_1(a,b) = &- a^2b - ab^2 - ab \\
p = 5 : f_1(a,b) = &- a^4b - 2a^3b^2 - 2a^2b^3 - ab^4 - 2a^3b + 2a^2b^2 \\
&- 2ab^3 - a^2b - ab^2 \\
p = 7 : f_1(a,b) = &- a^6b - 3a^5b^2 + 2a^4b^3 + 2a^3b^4 - 3a^2b^5 - ab^6 \\
&- 3a^5b + 3a^4b^2 - 3a^3b^3 + 3a^2b^4 - 3ab^5 + a^4b \\
&+ 2a^3b^2 + 2a^2b^3 + ab^4 - 3a^2b - 3ab^2 \\
p = 11 : f_1(a,b) = &- a^{10}b - 5a^9b^2 - 4a^8b^3 + 3a^7b^4 + 2a^6b^5 + 2a^5b^6 \\
&+ 3a^4b^7 - 4a^3b^8 - 5a^2b^9 - ab^10 - 5a^9b + 5a^8b^2 \\
&- 5a^7b^3 + 5a^6b^4 - 5a^5b^5 + 5a^4b^6 - 5a^3b^7 \\
&+ 5a^2b^8 - 5ab^9 - 2a^8b + 3a^7b^2 - 4a^6b^3 + 5a^5b^4 \\
&+ 5a^4b^5 - 4a^3b^6 + 3a^2b^7 - 2ab^8 - 4a^6b - a^5b^2 \\
&+ 2a^4b^3 + 2a^3b^4 - a^2b^5 - 4ab^6 - 5a^4b + a^3b^2 \\
&+ a^2b^3 - 5ab^4 - 4a^2b - 4ab^2
\end{aligned}$$

As the polynomial in its expanded form seems to have a degree of only p instead of the expected $2p-2$ as noted above, this would imply a theoretical best depth of $\lceil \log_2(p) \rceil$ instead of $\lceil \log_2(p-1)+1 \rceil$ from our closed formula. Thus, it seems natural to examine the effort for computing the polynomial in this expanded form to see if this might be more efficient. Since the computation of the coefficient

$$(p-1)^{-x} \cdot \sum_{i=1}^{p-1} \left(i^{-y} \cdot \sum_{j=1}^{i} j^{-x} \right)$$

for each term is not encrypted, it costs nearly nothing compared to the encrypted computations, so we ignore this cost. Also, we will distinguish between constant multiplication

[3]The Gaussian sum formula states that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for all $n \in \mathbb{N}$.

(i.e., multiplying the ciphertext by its plaintext coefficient) and regular field multiplication of two ciphertexts, as the former is often much more efficient than the latter. We also assume that constant multiplication does not increase the depth.
Since we implemented precomputation for the closed formula, we will do the same here:

1. Compute $a^2, \ldots, a^{p-1}$ and $b^2, \ldots, b^{p-1}$, where a^k is computed with minimum depth and only one multiplication from $a^{\lfloor k/2 \rfloor} \cdot a^{\lceil k/2 \rceil}$:
   - Field additions: 0
   - Field multiplications: $2p - 4$
   - Constant multiplications: 0
   - Multiplicative depth: At most $\lceil \log_2(p-1) \rceil$
2. Let nt be the number of terms in the expanded polynomial. Then for each of the nt terms of the form $\alpha \cdot a^x \cdot b^y$, multiply the precomputed factors a^x and b^y (1 field multiplication) and multiply the result by the plaintext coefficient α (1 constant multiplication):
   - Field additions: 0
   - Field multiplications: nt
   - Constant multiplications: nt
   - Multiplicative depth: $+1$
3. Sum up the nt terms:
   - Field additions: $nt - 1$
   - Field multiplications: 0
   - Constant multiplications: 0
   - Multiplicative depth: $+0$

As we can see, we now need to estimate the number of terms in the polynomial. To do this, we calculated the exact number of terms for all primes less than 350. Next, we ran a quadratic regression on the number of terms with respect to the prime, setting aside 10 values (see below) to check the estimate. The result, with an incredibly high correlation coefficient of 0.99998, is that the number of terms in the expanded polynomial is about

$$nt(p) := 0.249657916p^2 + 0.869559p + 3.1487.$$

The fit of the regression curve can be seen in Figure 2.1. We see that the curve fits the data extremely well – the values predicted by this formula compared to the actual values are shown in Table 2.1.

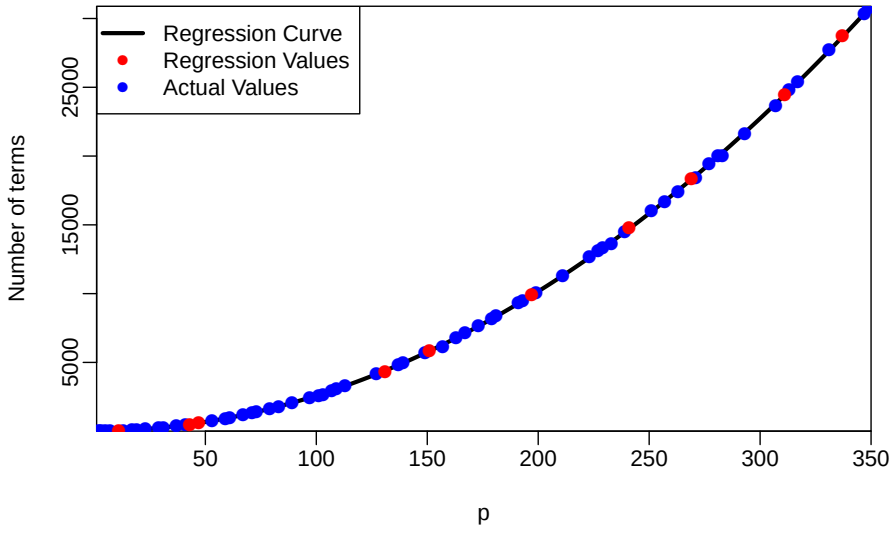

Figure 2.1: The number of terms for different p, the regression curve from these terms, and computed actual values.

p	11	43	47	131	151	197	241	269	311	337
predicted	43	502	596	4401	5827	9863	14713	18303	24421	28650
actual	39	503	597	4311	5849	9897	14759	18357	24471	28727

Table 2.1: Actual vs. estimated values for the number of terms in the expanded form of f_1 for the values of the test set.

Using this formula (rounded to $nt(p) \approx 0.25p^2 + 0.87p + 3.15$), we get as a total effort for the computation[4]:

Step	**Additions**	**Field Mults**	**Const Mults**	**Depth**
1	0	$2p-4$	0	$\lceil \log_2(p-1) \rceil$
2	0	nt	nt	$+1$
3	$nt-1$	0	0	$+0$
Total (nt)	$nt-1$	$2p+nt-4$	nt	$\lceil \log_2(p-1) \rceil + 1$ Theoretical best: $\lceil \log_2(p) \rceil$
Total	$0.25p^2$ $+0.87p+2.15$	$0.25p^2$ $+2.87p-0.85$	$0.25p^2$ $+0.87p+3.15$	$\lceil \log_2(p-1) \rceil + 1$ Theoretical best: $\lceil \log_2(p) \rceil$

We can see that as p increases, the closed formula

$$f_1 = \sum_{i=1}^{p-1} \left(l_i(b) \cdot \sum_{j=1}^{i} l_{p-j}(a) \right)$$

has much lower computation effort regarding the number of additions and multiplications, even when ignoring the constant multiplications in the expanded formula as we do. The expanded form does, however, promise a slightly lower depth, where the difference between the two forms is at most 1. Concretely, the number of operations to compute f_1 can be seen in Figure 2.2.

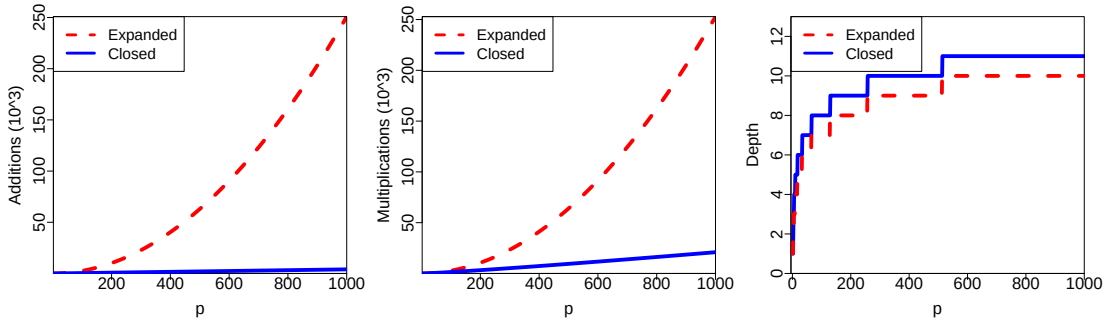

Figure 2.2: Number of field additions, field multiplications and multiplicative depth to compute f_1 through the closed and expanded forms (for the expanded form, the theoretical optimal depth is used).

Additionally, we would like to point out that if one were to implement this p-adic encoding, the closed formula can easily be realized by a loop, whereas it is questionable if one would actually want to implement the expanded form with e.g. nearly 10000 terms for $p = 197$. To be as unbiased as possible, we always took the better of the two values as an upper bound on the effort. Thus, we use the field addition and multiplication metric from the

[4]The theoretical best depth comes from the degree of the polynomial being p, and our precomputation may yield a depth one higher than this optimum. This is because we compute $a^x b^y = a^x \cdot b^y$, but the depth optimal way to compute the product might be some other form $(ab)^w \cdot a^v \cdot b^z$ with $x = w + v,\ y = w + z$.

closed formula, but the best possible depth of $\lceil \log_2(p) \rceil$ from the expanded formula in our following effort analysis.

2.3.1.4 *Summary: Effort for f_1*

In conclusion, the values we use for the computational effort of f_1 are:

Additions	**Multiplications**	**Depth**
$4p - 6$	$2p \cdot \log_2(p) + p - 2 \cdot \log_2(p) - 3$	$\lceil \log_2(p) \rceil$

2.3.2 Effort for f_2

Recall from Theorem 2.1 that $f_2(a, b) = (p-1) \cdot l_{p-1}(a+b)$ with

$$l_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{p-1} (x - j).$$

First, note again that we do not count the multiplication with $p-1$ as a multiplication, as this is a multiplication with a constant, which is usually cheaper than the multiplication of two ciphertexts. More importantly, multiplying with $p-1$ just switches the sign, so in schemes that support subtraction of two ciphertexts, we can rewrite

$$r_i = f_1(a_{i-1}, b_{i-1}) - r_{i-1} \cdot l_{p-1}(a_{i-1}, b_{i-1})$$

instead of the original

$$\begin{aligned} r_i &= f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1}) \\ &= f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot (p-1) \cdot l_{p-1}(a_{i-1}, b_{i-1}) \end{aligned}$$

and incur no additional effort at all, which is what we assume here.

This leaves us with the task of computing $l_{p-1}(a+b) = \prod_{0}^{p-2} (a+b-j)$, which is a product of $p-1$ factors:

1. We require one addition to compute $(a+b)$, and additional $p-2$ additions to compute $(a+b) - j$ for each $j \in \{1, \ldots, p-2\}$ (we need no computation for $j = 0$), which yields $p-1$ additions in total.
2. To multiply the $p-1$ factors, we need $p-2$ multiplications in total. Implementing the multiplication in way with the least depth (i.e., in a fanned-out fashion[5]) we obtain a depth of $\lceil \log_2(p-1) \rceil$.

So to compute f_2, we incur the following effort:

Step	**Additions**	**Multiplications**	**Depth**
1	$p-1$	0	0
2	0	$p-2$	$\lceil \log_2(p-1) \rceil$
Total	$p-1$	$p-2$	$\lceil \log_2(p-1) \rceil$

[5] What we mean by this is the balanced way of multiplying using something similar to a binary tree structure: For example, the product $a \cdot b \cdot c \cdot d \cdot e \cdot f$ would be computed as $\Big((a \cdot b) \cdot (c \cdot d)\Big) \cdot (e \cdot f)$ (with depth 3) rather than $(((((a \cdot b) \cdot c) \cdot d) \cdot e) \cdot f)$ of depth 5.

2.3.3 Total Effort

Putting these numbers together with the effort for f_2 and our analysis from the beginning of Section 2.3, we will shortly obtain the total cost for computing c_i. However, we first need to make some observations about the depth:

- Firstly, r_0 is 0, so we do not need to compute anything at all.
- Second, $r_1 = f_1(a_0, b_0)$ and thus automatically has the depth of f_1.
- For subsequent r_i, in the beginning of Section 2.3 we derived a depth of

$$d(c_i) = \max\{d(f_1), \max\{d(r_{i-1}), d(f_2)\} + 1\}.$$

- Using this formula for r_2, we get

$$\begin{aligned} d(r_2) &= \max\{d(f_1), \max\{d(r_1), d(f_2)\} + 1\} \\ &= \max\{d(f_1), d(f_2)\} + 1 \\ &= \max\{\lceil \log_2(p) \rceil, \lceil \log_2(p-1) \rceil\} + 1 \\ &= \lceil \log_2(p) \rceil + 1 \end{aligned}$$

- From here on, it is clear that $d(r_i) > d(f_1) \geq d(f_2)$, so the depth will increase by 1 with each i, leaving us with a total depth of

$$d(r_i) = \lceil \log_2(p) \rceil + i - 1.$$

Now, we can give the total cost for computing c_i (for $i > 1$):

	Additions	**Multiplications**	**Depth**
f_1	$4p - 6$	$2p \cdot \log_2(p) + p - 2 \cdot \log_2(p) - 3$	$\lceil \log_2(p) \rceil$
f_2	$p - 1$	$p - 2$	$\lceil \log_2(p-1) \rceil$
Total	$3 + \text{Adds}(f_1) + \text{Adds}(f_2)$	$1 + \text{Mults}(f_1) + \text{Mults}(f_2)$	$\lceil \log_2(p) \rceil + i - 1$
Total	$5p - 4$	$2p \cdot \log_2(p) + 2p - 2 \cdot \log_2(p) - 4$	$\lceil \log_2(p) \rceil + i - 1$

For completeness, we also give the effort for the two least significant and the most significant digits, as these deviate slightly from the above formulas. This should only have an impact for computations with very short lengths.
Special cases: The effort for

$$c_0 = a_0 + b_0$$

is only 1 field addition, and that for

$$c_1 = a_1 + b_1 + r_1 = a_1 + b_1 + f_1(a_0, b_0)$$

is $4p + 4$ additions, $2p \cdot \log_2(p) + p - 2 \cdot \log_2(p) + 1$ multiplications, and $\lceil \log_2(p) \rceil$ depth. Another special case is

$$c_{n+1} = r_{n+1},$$

which has 2 field additions less than the other $c_i, i > 1$.

Remark 1. Due to the unique structure when $p = 2$ (i.e., a, b, and r all have the same degree 1), there is a more efficient formula:

$$r_i = (a + b) \cdot (a + r_{i-1}) + a$$

which only needs 1 multiplication (see [BPP00]). However, this approach does not carry over when $p > 2$, as the carry formula is no longer symmetrical regarding r_{i-1}.

2.4 COST ANALYSIS FOR COMPUTING ON ENCRYPTED NATURAL NUMBERS

In this section, we analyze the effort required to add or multiply two natural numbers encoded p-adically using the polynomial we derived in Section 2.2.
In **Subsection 2.4.1**, we calculate the cost of adding two numbers in p-adic encoding by determining the number of digits required for the respective base, and then using the costs per digit from Section 2.3 to determine the total cost of addition. We also present **Theorem 2.3**, which tells us that $p = 2$ is the best choice in terms of field additions and multiplications, and that the optimal depth depends on the size of the number being encoded, or more specifically, the required number of digits.
Lastly, **Subsection 2.4.2** analyzes the cost of multiplying two numbers in p-adic encoding, using the addition from the previous subsection as a building block. We see that the cost analysis for addition carries over to multiplication and obtain $p = 2$ as the optimal choice regarding field additions and multiplications here as well, along with a variable optimum for depth.

2.4.1 The Cost of Adding Two Natural Numbers

Suppose we have some natural number x (in decimal representation). Then to represent x in p-adic encoding, we require $\lfloor \log_p(x) \rfloor + 1$ digits. Adding two such numbers, our result will have $\lfloor \log_p(x) \rfloor + 2$ digits. We consider two cases (due to the different effort for c_0):

- **Case 1:** $0 \leq x \leq p-1$. This means that our number can be encoded with one digit, and the result will have two digits. We have
  - $c_0 = a_0 + b_0$ with an effort of 1 addition, and
  - $c_1 = r_1 = f_1(a_0, b_0)$ with an effort of $4p-6$ additions, $2p \cdot \log_2(p) + p - 2 \cdot \log_2(p) - 3$ multiplications and a depth of $\lceil \log_2(p) \rceil$.

  In total, the cost of adding two 1-digit numbers is:

	Additions	**Multiplications**	**Depth**
c_0	1	0	0
c_1	$4p - 6$	$2p \cdot \log_2(p) + p - 2 \cdot \log_2(p) - 3$	$\lceil \log_2(p) \rceil$
Total	$4p - 5$	$2p \cdot \log_2(p) + p - 2 \cdot \log_2(p) - 3$	$\lceil \log_2(p) \rceil$

- **Case 2:** $p \leq x$. This means that x will be encoded with $2 \leq \ell := \lfloor \log_p(x) \rfloor + 1$ digits and the result will have $\ell + 1$ digits. We again have
  - $c_0 = a_0 + b_0$ with an effort of 1 addition.
  - $c_1 = a_1 + b_1 + r_1 = a_1 + b_1 + f_1(a_0, b_0)$ with an effort of $4p - 4$ additions, $2p \cdot \log_2(p) + p - 2 \cdot \log_2(p) - 3$ multiplications and a depth of $\lceil \log_2(p) \rceil$.
  - The last digit $c_\ell = r_\ell$ has a cost of $5p-6$ additions, $2p \cdot \log_2(p) + 2p - 2 \cdot \log_2(p) - 4$ multiplications, and a depth of $\lceil \log_2(p) \rceil + 1$.
  - The remaining $\ell - 2$ middle digits c_i have the normal effort of $5p - 4$ additions, $2p \cdot \log_2(p) + 2p - 2 \cdot \log_2(p) - 4$ multiplications and a depth of $\lceil \log_2(p) \rceil + i - 1$.

  In total, the cost of adding two ℓ-digit numbers, $\ell \geq 2$, is:

	Additions	**Multiplications**	**Depth**
c_0	1	0	0
c_1	$4p - 4$	$2p \cdot \log_2(p) + p$ $-2 \cdot \log_2(p) - 3$	$\lceil \log_2(p) \rceil$
c_ℓ	$5p - 6$	$2p \cdot \log_2(p) + 2p$ $-2 \cdot \log_2(p) - 4$	$\lceil \log_2(p) \rceil + \ell - 1$
c_i	$5p - 4$	$2p \cdot \log_2(p) + 2p$ $-2 \cdot \log_2(p) - 4$	$\lceil \log_2(p) \rceil + i - 1$
Total	$(5\ell - 1) \cdot p - (4\ell + 1)$	$2\ell \cdot p \cdot \log_2(p)$ $+(2\ell - 1) \cdot p$ $-2\ell \cdot \log_2(p)$ $-(4\ell - 1)$	$\lceil \log_2(p) \rceil + \ell - 1$

Keeping in mind that $\ell := \lfloor \log_p(x) \rfloor + 1$ in the general Case 2, we can now clearly see the main result of this section:

Theorem 2.3. *Asymptotically, the costs for adding two natural numbers in p-adic encoding increase as p increases.*

Proof: We can see from the above cases that while the required encoding length of a number x encoded in base p

$$\ell = \lfloor \log_p(x) \rfloor + 1 = \lfloor \frac{\log_2(x)}{\log_2(p)} \rfloor + 1 = \Theta(\frac{1}{\log_2(p)})$$

only decreases logarithmically, the effort grows with p as

$$\Theta(\ell \cdot p) = \Theta((\lfloor \frac{\log_2(x)}{\log_2(p)} \rfloor + 1) \cdot p) = \Theta(p + \frac{p}{\log_2(p)}) = \Theta(p)$$

for additions and as

$$\Theta(\ell \cdot p \cdot \log_2(p)) = \Theta((\lfloor \frac{\log_2(x)}{\log_2(p)} \rfloor + 1) \cdot p \cdot \log_2(p)) = \Theta(p \cdot \log_2(p) + p) = \Theta(p \cdot \log_2(p))$$

for multiplications. The depth

$$\lceil \log_2(p) \rceil + \ell - 1 = \lceil \log_2(p) \rceil + \lfloor \frac{\log_2(x)}{\log_2(p)} \rfloor$$

also increases logarithmically ($\Theta(\log_2(p))$). Thus, for all of our cost metrics it holds that they increase as p increases.

□

This theorem implies that the best choice is likely a very small prime like $p = 2$.
We would like to point out again that if the function being evaluated is known beforehand, choosing p so large that computations do not wrap around mod p is likely to be faster – however, this is not *Fully* Homomorphic Encryption but rather *Somewhat* Homomorphic Encryption (see Section 1.2.2). Theorem 2.3 holds for p-adic encoding used in true FHE. We have illustrated this Theorem through Figure 2.3, which shows the effort as p grows for selected values of x.

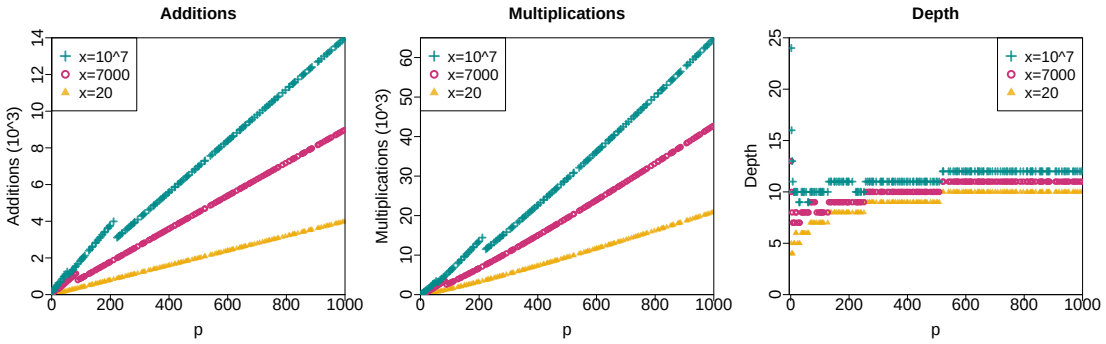

Figure 2.3: Number of field additions, multiplications and depth for adding $x = 20/7000/10^7$ to a number of same size. The horizontal axis is the encoding base p, the vertical axis is number of operations/depth, and the plots correspond to the three numbers.

We can see that indeed, the number of additions, multiplications and the depth increase significantly as the encoding base p increases. **We also see that for additions and multiplications, our result does not only hold asymptotically, but also for the concrete values: p=2 indeed has the lowest cost and is thus the best choice.** Note that the jags in the first two diagrams occur when the base prime becomes so large that one digit less is required for encoding than under the previous prime, so the effort drops briefly before increasing again.
The diagram for depth shows us an interesting phenomenon[6] that is hidden in the asymptotic analysis: For low primes, it is actually the required number of digits that dominates the total depth cost. This problem becomes more pronounced the larger the encoded number is, and vanishes after the first few primes as the expected asymptotic cost takes

[6]This phenomenon also occurs for the non-optimal depth $\lceil \log_2(p-1) \rceil + \ell$.

over. This means that if depth is the only cost metric that is being considered (where as we have explained before, we feel that as soon as bootstrapping is unavoidable, the total number of multiplications is the more important metric), choosing a slightly larger prime than 2 yields better results at the cost of significantly increased multiplications. Also, the optimal choice of p depends heavily on the numbers that are being encoded. For example, in Figure 2.3, the depth-optimal choices for adding x would be $p = 3$ for $x = 20$, $p = 7$ for $x = 7000$, and $p = 29$ for $x = 10^7$.

2.4.2 The Cost of Multiplying Two Natural Numbers

We now analyze the cost of multiplying two natural numbers

$$a \cdot b = (a_{\ell-1} \dots a_0) \cdot (b_{\ell-1} \dots b_0)$$

in p-adic encoding. We examine the standard multiplication algorithm because the advanced multiplication algorithms used (on unencrypted data) today usually have many IF-THEN instructions. Since doing this in encrypted form always requires executing the entire binary decision tree, we have not yet found an advanced algorithm that translates well to the encrypted setting, because most of these benefits disappear when computing the entire decision tree is necessary.
In performing this multiplication, there are two main steps:

- First, we perform a one digit multiplication of each b_i with all of $a_{\ell-1}a_{\ell-2} \dots a_1a_0$.
- In the second step, we add up the rows we obtained in this way (shifting one space to the left with each increasing row) using the addition from the previous subsection as a building block.

Definition 2.2 (Multiplication matrix)**.** The term *multiplication matrix* refers to the rows from the first step written beneath each other to form a matrix (with some blank entries), shifting one space to the left with each increasing row.

As an example, consider the multiplication of two 3-digit numbers:

$$\begin{array}{cccccccc} a_2 & a_1 & a_0 & \cdot & b_2 & b_1 & b_0 & \\ \hline & & & x_3 & x_2 & x_1 & x_0 & \\ & & y_3 & y_2 & y_1 & y_0 & & \\ & z_3 & z_2 & z_1 & z_0 & & & \\ \hline c_5 & c_4 & c_3 & c_2 & c_1 & c_0 & & \end{array}$$

The first step is obtaining the rows

$$x_3x_2x_1x_0 = a_2a_1a_0 \cdot b_0, \quad y_3y_2y_1y_0 = a_2a_1a_0 \cdot b_1 \quad \text{and} \quad z_3z_2z_1z_0 = a_2a_1a_0 \cdot b_2.$$

Except in the case of $p = 2$, where $b_i \in \{0, 1\}$, so

$$x_3x_2x_1x_0 = x_2x_1x_0 = (a_2 \cdot b_0)(a_1 \cdot b_0)(a_0 \cdot b_0),$$

this actually requires some computational effort. The multiplication matrix consists of the three rows $x_3x_2x_1x_0$, $y_3y_2y_1y_0$ and $z_3z_2z_1z_0$, and the second step consists of adding these rows to obtain the result $c_5c_4c_3c_2c_1c_0$.

2.4.2.1 *Computing Step 1*

In the case that $a_0 \cdot b_0 > p$, we have a carry into the next digit. Concretely, we write

$$a_i \cdot b_j + r_i = r_{i+1} \cdot p + x_i$$

with $x_i < p$, where $r_0 = 0$ and $r_i \in \{0, \ldots, p-2\}$.[7] Very similarly to the uniqueness proof of Theorem 2.1, we can obtain the formula for this carry digit through a 3-fold Lagrange approximation over the variables a_i, b_i and r_i as

$$\sum_{y=1}^{p-1} l_y(b) \cdot \sum_{i=1}^{y} l_i(a) \cdot \sum_{j=1}^{i} l_j(r).$$

This means that the formula for the carry r_i will be a triple sum over l_i-functions.
Note, however, that this polynomial is not unique, as it can take multiple values for $r = p - 1$. For example, although the degree would generally be expected as $3 \cdot (p-1)$, we have experimentally seen by generating these different polynomials that among them, there seems to be one which has a degree of only $2 \cdot (p-1) + 1$. Since we aim to always use the best possible choice for each cost metric, we will use this value as our lower bound on the depth in our effort analysis. For the number of additions and multiplications, we instead again use the closed Lagrangian formula as in the previous section, as they promise a lower cost for these metrics. Using precomputation and the time-memory tradeoff from Section 2.3.1.2, we obtain the effort for computing r_{i+1} from (a_i, b_i, r_i) this way through the following steps:

1. Compute $(a-j), (b-j)$ and $(r-j)$ for $j = 1, \ldots, p-1$ with $3p-3$ additions.
2. Compute $l_i(a), l_i(b), l_i(r)$ for $i = 1, \ldots, p-1$ via time-memory tradeoff using the precomputed factors from the previous step, which needs $3 \cdot (p \cdot \log_2(p) - \log_2(p) - 1)$ multiplications
3. Compute sums $\sum_{j=1}^{i} l_j(r)$ for all i, needing $p-2$ additions.
4. Compute $l_i(a) \cdot \sum_{j=1}^{i} l_j(r)$ for $i = 1, \ldots, p-1$, which incurs one multiplication per i.
5. Compute sums $\sum_{i=1}^{y} l_i(a) \cdot \sum_{j=1}^{i} l_j(r)$ for all y, needing $p-2$ additions.
6. Compute $l_i(b) \cdot \left(\sum_{i=1}^{y} l_i(a) \cdot \sum_{j=1}^{i} l_j(r) \right)$ for $i = 1, \ldots, p-1$, which needs $p-1$ multiplications.

[7] r_i cannot be $p-1$ because the maximum first carry r_1 happens at

$$(p-1) \cdot (p-1) = (p-2) \cdot p + 1,$$

so $r_1 \leq p-2$, and subsequently the maximum that can occur is at

$$a_i \cdot b_0 + r_i = (p-1) \cdot (p-1) + (p-2) = (p-2) \cdot p + (p-1),$$

so $r_i \leq p-2$.

7. Lastly, compute $\sum_{y=1}^{p-1} l_y(b) \cdot \sum_{i=1}^{y} l_i(a) \cdot \sum_{j=1}^{i} l_j(r)$, needing $p-2$ additions.

With these steps (with the values in parentheses denoting the depth from these steps, and the final depth as the lowest depth $\lceil \log_2(2p-1) \rceil$ for the experimentally found polynomial of lowest degree $2 \cdot (p-1)+1$ as mentioned above), we get a total effort for computing the carry of:

Step	**Field Additions**	**Field Multiplications**	**Depth**
1	$3p-3$	0	(0)
2	0	$3p \cdot \log_2(p) - 3 \cdot \log_2(p) - 3$	$\left(\lceil \log_2(p-1) \rceil \right)$
3	$p-2$	0	(0)
4	0	$p-1$	(+1)
5	$p-2$	0	(0)
6	0	$p-1$	(+1)
7	$p-2$	0	(0)
Total	$6p-9$	$3p \cdot \log_2(p) + 2p - 3 \cdot \log_2(p) - 5$	$\lceil \log_2(2p-1) \rceil$ $\left(\lceil \log_2(p-1) \rceil + 2 \right)$

Recalling that the digits are computed as $a_i \cdot b_j + r_i$, each digit (except the special first and last digit) has the effort of the carry and one additional addition and multiplication (which does not increase the depth). Thus, each row, which has length $\ell + 1$, roughly has as its effort (where we increase the depth by $\log_2$ of the degree, i.e., $\log_2(2p-1)$ with each i):

Digit	**Field Additions**	**Field Multiplications**	**Depth**
0	0	1	1
i	$6p-8$	$3p \cdot \log_2(p) + 2p - 3 \cdot \log_2(p) - 4$	$i \cdot \lceil \log_2(2p-1) \rceil$
ℓ	$6p-9$	$3p \cdot \log_2(p) + 2p - 3 \cdot \log_2(p) - 5$	$\ell \cdot \lceil \log_2(2p-1) \rceil$
Total	$\ell \cdot (6p-8) - 1$	$\ell \cdot (3p \cdot \log_2(p) + 2p - 3 \cdot \log_2(p) - 5)$	$\ell \cdot \lceil \log_2(2p-1) \rceil$

Doing this for all ℓ rows, the first of the two steps has the following effort:

	Field Additions	**Field Multiplications**	**Depth**
Total	$\ell^2 \cdot (6p-8) - \ell$	$\ell^2 \cdot (3p \cdot \log_2(p) + 2p - 3 \cdot \log_2(p) - 5)$	$\ell \cdot \lceil \log_2(2p-1) \rceil$

2.4.2.2 *Computing Step 2*

The second step consists of adding all the rows that we computed in the first step. At this point, we present a short optimization from [JA16] that saves some effort in this step:

Improving Multiplication:
Having computed the rows that we want to sum up, we can first apply a logarithmic approach (i.e., with rw_i denoting row i, we compute $(rw_1 + rw_2) + (rw_3 + rw_4)$ instead of $(((rw_1 + rw_2) + rw_3) + rw_4)$ to keep the involved lengths and thus effort as low as possible.

Secondly, it is noteworthy that we can save computation power by modifying the addition operation: As can easily be seen, we are always adding rows of different lengths. While the naïve approach of padding the right-hand side of the shorter number with 0's and applying normal addition would also work, we can save some effort by copying the excess digits of the longer number and then performing addition on the remaining shorter parts. Generally, this approach allows us to reduce the input length for the addition subroutine, which is an important factor in depth optimization.

Example 2.2: *As an example, suppose we are multiplying two* 4*-digit numbers:*

$$\begin{array}{cccccccccc} a_3 & a_2 & a_1 & a_0 & \cdot & b_3 & b_2 & b_1 & b_0 \\ \hline & & & & & w_4 & w_3 & w_2 & w_1 & w_0 \\ & & & x_4 & x_3 & x_2 & x_1 & x_0 & & \\ & & y_4 & y_3 & y_2 & y_1 & y_0 & & & \\ z_4 & z_3 & z_2 & z_1 & z_0 & & & & & \end{array}$$

Then the naive way of adding rows 1 *and* 2 *would be to pad row* 2 *(note that* w_4 *and* x_4 *are at most* 1*, so there is no carry to form a potential* c_6*):*

$$\begin{array}{cccccccc} & & & w_4 & w_3 & w_2 & w_1 & w_0 \\ + & & x_4 & x_3 & x_2 & x_1 & x_0 & 0 \\ \hline = & & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \end{array}$$

This would mean adding numbers of lengths 5 *and* 6 *together. However, if we just copy* w_0 *to the result line, we reduce the length:*

$$\begin{array}{cccccccc} & & & w_4 & w_3 & w_2 & w_1 & \vdots w_0 \\ + & & x_4 & x_3 & x_2 & x_1 & x_0 & \vdots \\ \hline = & & c_5 & c_4 & c_3 & c_2 & c_1 & \vdots w_0 \end{array}$$

Now we only need to add an number of length 4 *to a number of length* 5*. In the next step (suppose that* $d_6 d_5 \dots d_1 d_0$ *is the result of adding rows* 3 *and* 4 *in the same way), the effect is even more pronounced:*

$$\begin{array}{cccccccccc} & & & & c_5 & c_4 & c_3 & c_2 & \vdots c_1 & c_0 \\ + & & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 & \vdots & \\ \hline = & & w_7 & w_6 & w_5 & w_4 & w_3 & w_2 & \vdots c_1 & c_0 \end{array}$$

Here, padding the lower row would lead to input lengths 6 *and* 8*, whereas our optimization reduces them to* 4 *and* 6*.*
We see that even with such small numbers with input length 4*, we have reduced input lengths to the addition routine by* 4 *in total, and this effect scales with larger inputs, so we can indeed save some effort this way.* ———

Applying this optimization to our scenario (and treating the addition of numbers of different lengths like the addition of two numbers of the larger length), we first do $\frac{\ell}{2}$ additions

with $\ell+1$ digits (because the rightmost one is copied). Next, we do $\frac{\ell}{4}$ additions with $\ell+3$ digits, because the rightmost two are copied down. We continue this until we have only one row left, with $\texttt{Add}(x)$ denoting the respective effort for adding two number of x digits as computed in the previous subsection.
For the depth, note from the table for Step 1 that the i^{th} digit in each row has depth $i \cdot \lceil \log_2(2p-1) \rceil$. Since the depth of the carry grows by 1 with each i in addition, but the next input has a depth $\lceil \log_2(2p-1) \rceil$ larger, the carry will have at most the depth of the upper row of the input (i.e., in $c_i = a_i + b_i + r_i$, the depth of r_i will be at most that of $\max\{d(a_i), d(b_i)\}$). This means that the depth does actually not increase through the addition, and the maximum depth remains at $\ell \cdot \lceil \log_2(2p-1) \rceil$.
Thus, we get as total cost of this second step:

Field Additions	**Field Multiplications**	**Depth**
$\sum_{i=1}^{\lceil \log_2(\ell) \rceil} \frac{\ell}{2^i} \cdot \texttt{Add}(\ell + 2^{i-1} + 1)$	$\sum_{i=1}^{\lceil \log_2(\ell) \rceil} \frac{\ell}{2^i} \cdot \texttt{Add}(\ell + 2^{i-1} + 1)$	$\ell \cdot \lceil \log_2(2p-1) \rceil$

2.4.2.3 *Total Multiplication Effort*

Putting the two steps together, we obtain as the total cost for multiplication:

Step	**Field Additions**	**Field Multiplications**	**Depth**
1	$\ell^2 \cdot (6p-8) - \ell$	$\ell^2 \cdot (3p \cdot \log_2(p) + 2p - 3 \cdot \log_2(p) - 5)$	$\ell \cdot \lceil \log_2(2p-1) \rceil$
2	$\sum_{i=1}^{\lceil \log_2(\ell) \rceil} \frac{\ell}{2^i} \cdot \texttt{Add}(\ell + 2^{i-1} + 1)$	$\sum_{i=1}^{\lceil \log_2(\ell) \rceil} \frac{\ell}{2^i} \cdot \texttt{Add}(\ell + 2^{i-1} + 1)$	No increase
Total	$\ell^2 \cdot (6p-8) - \ell + \sum_{i=1}^{\lceil \log_2(\ell) \rceil} \frac{\ell}{2^i} \cdot \texttt{Adds}(\ell + 2^{i-1} + 1)$	$\ell^2 \cdot (3p \cdot \log_2(p) + 2p - 3 \cdot \log_2(p) - 5) + \sum_{i=1}^{\lceil \log_2(\ell) \rceil} \frac{\ell}{2^i} \cdot \texttt{Adds}(\ell + 2^{i-1} + 1)$	$\ell \cdot \lceil \log_2(2p-1) \rceil$

Since this is a complex formula, we have plotted the costs for different inputs in Figure 2.4. We can see that for additions and multiplications, the effort is lowest at $p = 2$ and grows with increasing p, though there are again some sharp drops when the required number of digits decreases. As one would expect, the issue with depth has propagated from addition, which we used as a building block in multiplication: The best depth for $x = 20$ is obtained for $p = 23$, the best depth for $x = 7000$ is obtained for $p = 89$, and the best depth for $x = 10^7$ is obtained for $p = 223$. We would like to point out that these values are not the same values that were optimal for addition (e.g., $p = 89$ is far from optimal for adding $x = 7000$) – thus, if one were to use depth as the sole metric, the optimal choice of p not only depends on the size of the numbers one is working with, but also on the number of additions vs. multiplications one wants to perform on these inputs. In the context of outsourced information, it is also important to note that optimizing the choice of p in this way could leak unwanted information about the function that was applied to the data, which may be the computing party's business secret.

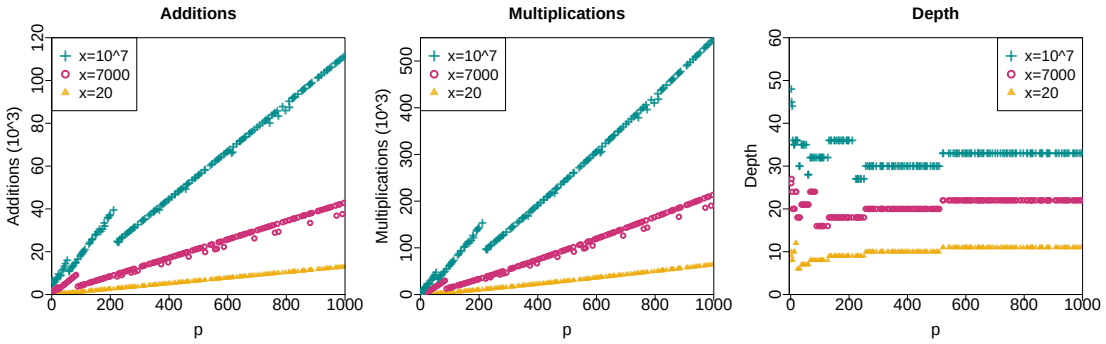

Figure 2.4: Number of field additions, field multiplications and multiplicative depth for multiplying $x = 20/7000/10^7$ to a number of same size. The horizontal axis is the encoding base p, the vertical axis is number of operations/depth, and the plots correspond to the three numbers.

2.5 EXTENSION TO ARBITRARY FINITE FIELDS

We now generalize our analysis to arbitrary finite fields as encoding bases. Much in the same way as in Sections 2.3 and 2.4, we have also analyzed the effort incurred when using $GF(p^k)$ for a prime p and a $k > 1$ as an encoding base. To this end, we first show how to generalize p-adic encoding to these more general fields in **Section 2.5.1**. We then compute the effort for computing each digit when adding two numbers in **Section 2.5.2**, and examine the effort for the total addition of two numbers in **Section 2.5.3**.

2.5.1 Encoding for $GF(p^k)$

First, recall that

$$GF(p^k) \cong \mathbb{Z}_p[X]/(f(x))$$

with $f(x)$ irreducible of degree k. We embed a decimal number between 0 and $p^k - 1$ into $GF(p^k)$, whose elements are polynomials over $\mathbb{Z}_p$ of degree less than k, through the insertion homomorphism: The element

$$a = \sum_{i=0}^{k-1} \alpha_i X^i \in GF(p^k) \quad (\text{with } \alpha_i \in \{0, 1, \ldots, p-1\})$$

encodes the number

$$\tilde{a} = \sum_{i=0}^{k-1} \alpha_i \cdot p^i \in \mathbb{N}.$$

Generalizing this to numbers larger than $p^k - 1$ is straightforward: We will represent a natural number $\tilde{a}$ as $a_n a_{n-1} \ldots a_1 a_0$, where $a_j \in GF(p^k)$, through

$$\tilde{a} = \sum_{j=0}^{n} \tilde{a_j} \cdot (p^k)^j,$$

with $\tilde{a_j}$ the natural number in $\{0, \ldots, p-1\}$ encoded by $a_j \in GF(p^k)$ as above.

Example 2.3: *Suppose we are working over* $GF(2^3) \cong \mathbb{Z}_2[X]/(X^3 + X + 1)$.
Then the single element $X^2 + 1 \in GF(2^3)$ *encodes the natural number* $2^2 + 1 = 5$*, whereas the single element* $X \in GF(2^3)$ *encodes the natural number* 2.
Using this structure as an encoding base, we can display natural numbers that are larger than $2^3 = 8$ *through tupels of* $GF(2^3)$*-elements. For example, the tupel*

$$(X + 1, 1, X^2)$$

encodes the number

$$(X + 1) \cdot (2^3)^2 + 1 \cdot (2^3)^1 + X^2 \cdot (2^3)^0 \mapsto 3 \cdot 8^2 + 1 \cdot 8 + 4 \cdot 1 = 204.$$

Encoding in the other direction works similarly:
Suppose we want to encode the number 8008 *in this fashion, then we first write it a sum of powers of* 2^3*:*

$$\begin{aligned} 8008 = 4096 + 3584 + 320 + 8 &= 1 \cdot 8^4 + 7 \cdot 8^3 + 5 \cdot 8^2 + 1 \cdot 8^1 + 0 \cdot 8^0 \\ &\mapsto 1 \cdot 8^4 + (X^2 + X + 1) \cdot 8^3 + (X^2 + 1) \cdot 8^2 + 1 \cdot 8^1 + 0 \cdot 8^0 \end{aligned}$$

which yields a tuple of

$$(1, X^2 + X + 1, X^2 + 1, 1, 0).$$

2.5.2 Effort per Digit

Having determined the appropriate encoding, we will now analyze the effort of adding two natural numbers in this encoding. Intuitively, we do not expect this to perform better than the encoding through $\mathbb{Z}_p$: The carry bit formula should roughly have the same effort as for $\mathbb{Z}_{p'}$ with p' of size comparable to p^k, but the addition is now more complicated. Concretely, the native addition structure of $GF(p^k)$ is that of $(\mathbb{Z}_p)^k$, i.e., it is done component-wise with no carry-over into other components, whereas we would need the addition of $\mathbb{Z}_{p^k}$ to natively support our encoding. Thus, we must emulate the addition $c_i = a_i + b_i + r_i$ in the same way as we compute the carry bit, so we expect a significantly larger effort here compared to $\mathbb{Z}_{p'}$ with $p' \approx p^k$.
The task at hand is now to compute $r_i = f(a_{i-1}, b_{i-1}, r_{i-1})$ and

$$c_i = \tilde{f}(a_i, b_i, r_i) := a_i +_{\mathbb{Z}_{p^k}} b_i +_{\mathbb{Z}_{p^k}} r_i.$$

We do this as before through bilinear Lagrangian Interpolation.

2.5.2.1 *Computing the Carry* r_i

We first note that as before, the carry can never be more than 1: Since the first carry r_0 is 0, the maximum value that can be reached in this step is for $a_0 = b_0 = p^k - 1$, yielding a result of $2p^k - 2 < 2p^k$, so the carry r_1 is at most 1. Similarly, with a carry r_i of at most 1 and a maximumum value of $a_i = b_i = p^k - 1$, we get a maximum of $a_i + b_i + r_i = 2p^k - 1 < 2p^k$ for the subsequent positions, so the carry is always at most 1. We then compute the function returning r_i as before by setting

$$r_i = f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1})$$

where (with all additions over $\mathbb{N}$)

$$f_1(a,b) = \begin{cases} 0, & a+b \leq p^k - 1 \\ 1, & a+b \geq p^k \end{cases} \text{ and } f_2(a,b) = \begin{cases} 1, & a+b = p^k - 1 \\ 0, & \text{else} \end{cases} \tag{2.21}$$

The l_i-functions are defined over these fields as follows:

$$l_i(x) = \prod_{\substack{j \in GF(p^k) \\ j \neq i}} (x - j).$$

Since we again have

$$f_2(a,b) = (p-1) \cdot l_{p-1}(a+b),$$

the function f_2 has a degree of $p^k - 1$, which is also the true degree (i.e., the higher-order terms do not generally vanish) and can be computed with $p^k - 2$ multiplications (for $p^k - 1$ factors, not counting the multiplication with $p - 1$ because it is a constant and can be implemented by subtraction) and $p^k - 1$ additions (one for $a + b$, and one for each factor of l_{p-1} except $(a + b - 0)$). Thus, for f_2 we have an effort of

Field Additions	**Field Multiplications**	**Depth**
$p^k - 1$	$p^k - 2$	$\lceil \log_2(p^k - 1) \rceil$

The other part, f_1, is obtained by Lagrangian Interpolation over the variables a and b, so the closed formula we obtain this way implies a theoretical degree of $2 \cdot p^k - 2$. Concretely,

$$f_1(a,b) = \sum_{j \in GF(p^k)^*} \left(l_j(b) \cdot \sum_{i \in GF(p^k)^*} l_i(a) \cdot \tilde{r}(i,j) \right) \tag{2.22}$$

where

$$\tilde{r}(i,j) = \begin{cases} 0, & i +_{\mathbb{N}} j < p^k \\ 1, & i +_{\mathbb{N}} j \geq p^k \end{cases}$$

is supplied as a known value in Lagrangian interpolation[8].

Remark 2. In Equation 2.22, we only sum over the non-zero elements of $GF(p^k)$ because if either i or j is 0, the carry can never be 1 and thus $\tilde{r}(i,j) = 0$. Of course, technically we could reduce the number of terms even further by only summing up the elements where $\tilde{r}(i,j) = 1$ as we did in the case of $\mathbb{Z}_p$, where the inner sum runs from $j = 1$ only to k rather than $p - 1$ in Theorem 2.1 for exactly this reason. However, due to the more abstract nature of $GF(p^k)$ and its non-trivial mapping to $\mathbb{N}$, it is very complex to express these non-zero terms in form of an iterator over which we can sum. For this reason, we will disregard this option since it makes only little difference: Using precomputation again, the only change is that the total number of ciphertext additions would decrease slightly. We will, however, not count the multiplication with $\tilde{r}(i,j)$ as a multiplication since it is always either 0 or 1, so we always either add the term or don't.

[8] Recall that we can, of course, easily compute these values in the clear when coming up with the formula for f_1. Later, when computing on the encrypted values, f_1 will do exactly the same thing as $\tilde{r}$, but expressed as a polynomial over $GF(p^k)$ which we can evaluate on encrypted inputs.

In reality, we again have the effect that the expanded formula (obtained by multiplying out the closed formula in Equation 2.22) seems to have a lower degree. Concretely, Table 2.2 shows the actual degree of f_1 for different values of p and k and the expected value of $2 \cdot p^k - 2$. The entry for $GF(7^4)$ is missing because merely computing the expanded form from the closed form (using Sage for the finite field arithmetic, without any encryption involved) would take roughly half a year on our computer.

$p\backslash k$	**1**	**2**	**3**	**4**
2	2	5	11	23
3	3	13	43	133
5	5	41	221	1121
7	7	85	631	x

$p\backslash k$	**1**	**2**	**3**	**4**
2	2	6	14	30
3	4	16	52	160
5	8	48	248	1248
7	12	96	684	4800

Table 2.2: Actual degree of f_1 vs. $2p^k - 2$

We can see that the degree of f_1 seems to be

$$p^k + (p-1) \cdot p^{k-1} - (p-1) = 2 \cdot p^k - p^{k-1} - (p-1).$$

Since this is asymptotically the same as the closed formula but would incur significant effort in computing a very large number of terms in the expanded version, we will stick with the closed formula in our analysis. As in Section 2.3.1, we use precomputation to compute intermediate results that are shared between the different l_i's in Step 2 of the following computation, where the notation L_v^i denotes that we have divided the p^k factors into v roughly equal sets, and this is the i^{th} of these sets. Then computing all such sets for $v = \frac{p^k}{2}, \ldots, 2$ requires a total of $p^k - 2$ multiplications similar to the explanation in Section 2.3.1, and computing l_i from these L_v takes $\log_2(p^k) - 1$ per l_i, of which there are $p^k - 1$. Remembering to do these computations for a and b each, we can compute all the l_i in Step 2 with

$$2 \cdot \Big(p^k - 2 + (p^k - 1) \cdot (\log_2(p^k) - 1)\Big) = 2p^k \cdot \log_2(p^k) - 2\log_2(p^k) - 2$$

multiplications. The complete computation is as follows:

1. Compute $(a - j)$ and $(b - j)$ for $j \in GF(p^k)^*$:
   - Field additions: $2p^k - 2$
   - Field multiplications: 0
   - Multiplicative depth: 0
2. Compute $l_i(a), l_i(b)$ for $i \in GF(p^k)^*$ with precomputation:
   - Field additions: 0
   - Field multiplications: $2p^k \cdot \log_2(p^k) - 2\log_2(p^k) - 2$
   - Multiplicative depth: $\lceil \log_2(p^k - 1) \rceil$ for each $l_i(a)$ and $l_i(b)$

3. Compute $l_j(b) \cdot \sum_{i \in GF(p^k)^*} l_i(a) \cdot \tilde{r}(i,j)$ for each $j \in GF(p^k)^*$ (recalling that we don't count the multiplication with $\tilde{r}(i,j)$ as a multiplication because this value is always in $\{0,1\}$ and not encrypted) with effort:
   - Field additions: $(p^k - 1) \cdot (p^k - 2) = p^{2k} - 3p^k + 2$
   - Field multiplications: $p^k - 1$
   - Multiplicative depth: $+1$
4. Lastly, sum up all the $l_j(b) \cdot \sum_{i \in GF(p^k)^*} l_i(a) \cdot \tilde{r}(i,j)$ to obtain
$$\sum_{j \in GF(p^k)^*} \left(l_j(b) \cdot \sum_{i \in GF(p^k)^*} l_i(a) \cdot \tilde{r}(i,j) \right) :$$
   - Field additions: $p^k - 2$
   - Field multiplications: 0
   - Multiplicative depth: $+0$

In total, to compute f_1 we get an effort of:

Step	**Field Additions**	**Field Multiplications**	**Depth**
1	$2p^k - 2$	0	0
2	0	$2p^k \cdot \log_2(p^k) - 2\log_2(p^k) - 2$	$\lceil \log_2(p^k - 1) \rceil$
3	$p^{2k} - 3p^k + 2$	$p^k - 1$	$+1$
4	$p^k - 2$	0	$+0$
Total	$p^{2k} - 2$	$2p^k \cdot \log_2(p^k) + p^k - 2\log_2(p^k) - 3$	$\lceil \log_2(p^k - 1) \rceil + 1$

And thus, combined with the effort for f_2, we get the costs of computing the carry r_i for the encoding base $GF(p^k)$ with $k > 1$ as:

	Field Additions	**Field Multiplications**	**Depth**
f_1	$p^{2k} - 2$	$2p^k \cdot \log_2(p^k) + p^k - 2\log_2(p^k) - 3$	$\lceil \log_2(p^k - 1) \rceil + 1$
f_2	$p^k - 1$	$p^k - 2$	$\lceil \log_2(p^k - 1) \rceil$
Total	$p^{2k} + p^k - 3$	$2p^k \cdot \log_2(p^k) + 2p^k - 2\log_2(p^k) - 5$	$\max\{\lceil \log_2(p^k - 1) \rceil + 1, d(r_{i-1})\} + 1$

2.5.2.2 *Computing the Addition*

As mentioned above, computing c_i is not as easy as simply computing $a_i + b_i + r_i$ in the field $GF(p^k)$, as the addition we require corresponds to the different structure $\mathbb{Z}_{p^k}$. Thus, we must also compute the function expressing this alien addition through Lagrangian interpolation. Concretely, we write

$$c_i = \tilde{f}(a_i, b_i, r_i) := a_i +_{\mathbb{Z}_{p^k}} b_i +_{\mathbb{Z}_{p^k}} r_i = (1 - r_i) \cdot \tilde{f}_1(a,_i , b_i) +_{GF(p^k)} r_i \cdot \tilde{f}_2(a_i, b_i)$$

with

$$\tilde{f}_1(a, b) = a +_{\mathbb{Z}_{p^k}} b \qquad \text{and} \qquad \tilde{f}_2(a, b) = a +_{\mathbb{Z}_{p^k}} b +_{\mathbb{Z}_{p^k}} 1. \tag{2.23}$$

We can do this because the carry r_i is at most 1, so we can easily split the function in this way. Similarly to above, both of these functions are obtained through a double interpolation over a and b and thus have a theoretical degree of $2 \cdot p^k - 2$ (except when $k = 1$, in this case we use the native addition and thus have degree 0). Since we are interested in how the degree propagates, it makes sense to rearrange the terms into

$$c_i = g(a_i, b_i, r_i) := a_i +_{\mathbb{Z}_{p^k}} b_i +_{\mathbb{Z}_{p^k}} r_i = g_1(a_i, b_i) +_{GF(p^k)} \ r_i \cdot g_2(a_i, b_i)$$

with

$$g_1(a, b) := \tilde{f}_1(a, b) \qquad \text{and} \qquad g_2(a, b) := \tilde{f}_2(a, b) - \tilde{f}_1(a, b) \tag{2.24}$$

The actual degrees for each and the expected value $2 \cdot p^k - 2$ can be found in Table 2.3, with the actual values for $GF(7^4)$ again missing due to enormous runtimes in computing the expanded forula from the closed one.

$p\backslash k$	**2**	**3**	**4**
2	4	10	22
3	9	39	129
5	25	205	1105
7	49	595	x

$p\backslash k$	**2**	**3**	**4**
2	2	6	14
3	6	24	78
5	20	120	620
7	42	336	x

$p\backslash k$	**2**	**3**	**4**
2	6	14	30
3	16	52	160
5	48	248	1248
7	96	684	4800

Table 2.3: Degrees of g_1 (left) and g_2 (middle) and the theoretical value (right).

The rule, which we checked with some additional values of p^k not in the table, appears to be:

$$\deg(g_1(a, b)) = 2p^k - p^{k-1} - p^2 + p \tag{2.25}$$

and

$$\deg(g_2(a, b)) = p^k - p. \tag{2.26}$$

Since this is relatively close to the degree $2 \cdot p^k - 2$ from the closed formula (and would again incur significantly more effort in working with the expanded form), we will use the closed formula and compute g_1 and g_2 using Equation 2.24. Thus, we first compute the effort of $\tilde{f}_1$ and $\tilde{f}_2$, which have similar costs and also constitute the cost for g_1, and then get the effort for g_2 with one additional addition. The effort analysis is very similar to the computation of f_1 above, except that we also compute $l_0(a)$ and $l_0(b)$ in Step 2 (which needs $2p^k \cdot \log_2(p^k) - 4$ multiplications instead of $2p^k \cdot \log_2(p^k) - 2\log_2(p^k) - 2$), keep track of the constant multiplications in Step 3, and sum over all of $GF(p^k)$ in Steps 3 and 4.

Thus, for the effort of computing $\tilde{f}_1$ and $\tilde{f}_2$ each, we get:

1. Compute $(a - j)$ and $(b - j)$ for $j \in GF(p^k)^*$:
   - Field additions: $2p^k - 2$
   - Field multiplications: 0
   - Multiplicative depth: 0
2. Compute $l_i(a), l_i(b)$ for $i \in GF(p^k)$ with precomputation as above:
   - Field additions: 0
   - Field multiplications: $2p^k \cdot \log_2(p^k) - 4$
   - Multiplicative depth: $\lceil \log_2(p^k - 1) \rceil$ for each $l_i(a)$ and $l_i(b)$
3. Compute $l_j(b) \cdot \sum_{i \in GF(p^k)} l_i(a) \cdot \tilde{v}(i, j)$ for each $j \in GF(p^k)$ (where $\tilde{v}(i, j)$ is the known cleartext value $i +_{\mathbb{Z}_{p^k}} j$, or $i +_{\mathbb{Z}_{p^k}} j +_{\mathbb{Z}_{p^k}} 1$ respectively, incurring a constant multiplication) with effort:
   - Field additions: $p^k \cdot (p^k - 1) = p^{2k} - p^k$
   - Field multiplications: p^k
   - Constant multiplications: p^{2k}
   - Multiplicative depth: $+1$
4. Lastly, sum up all the $l_j(b) \cdot \sum_{i \in GF(p^k)} l_i(a) \cdot \tilde{v}(i, j)$ to obtain
$$\sum_{j \in GF(p^k)} \left(l_j(b) \cdot \sum_{i \in GF(p^k)} l_i(a) \cdot \tilde{v}(i, j) \right) :$$
   - Field additions: $p^k - 1$
   - Field multiplications: 0
   - Multiplicative depth: $+0$

Thus, computing $\tilde{f}_1, \tilde{f}_2$ and g_1 each has a total effort of

Step	**Field Additions**	**Field Multiplications**	**Constant Multiplications**	**Depth**
1	$2p^k - 2$	0	0	0
2	0	$2p^k \cdot \log_2(p^k) - 4$	0	$\lceil \log_2(p^k - 1) \rceil$
3	$p^{2k} - p^k$	p^k	p^{2k}	$+1$
4	$p^k - 1$	0	0	$+0$
Total	$p^{2k} + 2p^k - 3$	$2p^k \cdot \log_2(p^k) + p^k - 4$	p^{2k}	$\lceil \log_2(p^k - 1) \rceil + 1$

and g_2 has the same effort, except one additional addition (so $p^{2k} + 2p^k - 2$ in total).

2.5.2.3 *Computing c_i*

Putting the results from this section together, we get (with all operations in $GF(p^k)$):

$$\begin{aligned} c_i(a_i, b_i, r_{i-1}) &= g_1(a_i, b_i) + r_i \cdot g_2(a_i, b_i) \\ &= g_1(a_i, b_i) + \big(f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1})\big) \cdot g_2(a_i, b_i) \end{aligned} \tag{2.27}$$

Denoting the multiplications with constant (i.e., unencrypted) values as `CMults`, we can thus see that the total effort for the number of operations is:

- Field additions: $\texttt{Adds}(g_1) + \texttt{Adds}(f_1) + \texttt{Adds}(f_2) + \texttt{Adds}(g_2) + 2$
- Field multiplications: $\texttt{Mults}(g_1) + \texttt{Mults}(f_1) + \texttt{Mults}(f_2) + \texttt{Mults}(g_2) + 2$
- Constant multiplications: $\texttt{CMults}(g_1) + \texttt{CMults}(f_1) + \texttt{CMults}(f_2) + \texttt{CMults}(g_2)$

However, we note that in computing $\tilde{f}_1$ and $\tilde{f}_2$ when computing g_1 and g_2, the first two steps are the same (computing the l_i's), and thus if we save the results, we only have to do this once. In fact, these are also the same two steps as in the computation of f_1 from the carry computation in the next digit (as f_1 is run on the inputs a_{i-1}, b_{i-1} in round i). The effort for these two steps is $2p^k - 2$ additions and $2p^k \cdot \log_2(p^k) - 4$ multiplications ($2p^k \cdot \log_2(p^k) - 2\log_2(p^k) - 2$ for f_1), so we will subtract these values in the following table, denoting this with "Redundancy":

	Field Additions	**Field Multiplications**	**Constant Multiplications**
f_1	$p^{2k} - 2$	$2p^k \cdot \log_2(p^k) + p^k - 2\log_2(p^k) - 3$	0
f_2	$p^k - 1$	$p^k - 2$	0
g_1	$p^{2k} + 2p^k - 3$	$2p^k \cdot \log_2(p^k) + p^k - 4$	p^{2k}
g_2	$p^{2k} + 2p^k - 2$	$2p^k \cdot \log_2(p^k) + p^k - 4$	p^{2k}
Additional	2	2	0
Redundancy	$-4p^k + 4$	$-4p^k \cdot \log_2(p^k) + 2\log_2(p^k) + 6$	0
Total	$3p^{2k} + p^k - 2$	$2p^k \cdot \log_2(p^k) + 4p^k - 5$	$2p^{2k}$

Regarding depth, we have

$$\text{Depth} = \max\{d(g_1), \max\{\max\{d(f_1), \max\{d(r_{i-1}), d(f_2)\} + 1\}, d(g_2)\} + 1\}$$

from Equation 2.27. Generally, the degree of r_{i-1} will be highest in that equation, and if minimal depth is our main objective, we can compute the term involving r_{i-1} as

$$r_{i-1} \cdot (f_2(a_{i-1}, b_{i-1}) \cdot g_2(a_i, b_i)),$$

increasing the degree by only one in each round. However, this will increase total computation because we still need to compute

$$r_i = \big(f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1})\big),$$

as it is an input to the next round. Still, we use the smaller value in our depth analysis, which can be found in Section 2.5.3.

2.5.3 Adding Natural Numbers

Using the results from the previous subsection, we calculate the total effort required to add two natural numbers $x \approx y$ of the same length $\ell = \lfloor \log_{p^k}(x) \rfloor + 1$. As before, we look at the special cases c_0 and c_1, c_2 and the last digit $c_\ell = r_\ell$ as well as the "regular" middle digits.

- $c_0 = a_0 +_{\mathbb{Z}_{p^k}} b_0 = g_1(a_0, b_0)$:
  - Field additions: $p^{2k} + 2p^k - 3$
  - Field multiplications: $2p^k \cdot \log_2(p^k) + p^k - 4$
  - Constant multiplications: p^{2k}
  - Multiplicative depth: $\lceil \log_2(p^k - 1) \rceil + 1$
- $c_1 = g_1(a_1, b_1) + f_1(a_0, b_0) \cdot g_2(a_1, b_1)$ (as $r_0 = 0$):
  - Field additions: $3p^{2k} - 2$
  - Field multiplications: $2p^k \cdot \log_2(p^k) + 3p^k - 4$
  - Constant multiplications: $2p^{2k}$
  - Multiplicative depth: $\lceil \log_2(p^k - 1) \rceil + 2$
- $c_2 = g_1(a_2, b_2) + \big(f_1(a_1, b_1) + r_1 \cdot f_2(a_1, b_1)\big) \cdot g_2(a_2, b_2)$: This entry is merely special in terms of depth, all other values are the same as the following c_i. Note that $r_1 = f_1(a_0, b_0)$, so its depth is $\lceil \log_2(p^k - 1) \rceil + 1$. This is the same as the depth of g_2 and one more than that of f_2, so computing $r_1 \cdot f_2 \cdot g_2$ will yield a depth of $d(c_2) = \lceil \log_2(p^k - 1) \rceil + 3$. Note that r_2 only has a degree of $\lceil \log_2(p^k - 1) \rceil + 2$, so it will have no impact in reality.
- $c_i = g_1(a_i, b_i) + r_i \cdot g_2(a_i, b_i)$ $(2 < i < l)$
  $= g_1(a_i, b_i) + \big(f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1})\big) \cdot g_2(a_i, b_i)$:
  As mentioned in the previous subsection, by expanding the formula and multiplying the term $r_{i-1} \cdot (f_2(a_{i-1}, b_{i-1}) \cdot g_2(a_i, b_i))$ in the appropriate order, the depth will only increase by 1 with each increase in i, and we will have $d(r_i) = d(c_i)$ (at the cost of increased computation). Thus, we get as minimum effort:
  - Field additions: $3p^{2k} + p^k - 2$
  - Field multiplications: $2p^k \cdot \log_2(p^k) + 4p^k - 5$
  - Constant multiplications: $2p^{2k}$
  - Multiplicative depth: $\lceil \log_2(p^k - 1) \rceil + i$
- $c_\ell = r_\ell = f_1(a_{\ell-1}, b_{\ell-1}) + r_{\ell-1} \cdot f_2(a_{\ell-1}, b_{\ell-1})$:
  Since we already computed the $l_i(a_{\ell-1})$ and $l_i(b_{\ell-1})$ in the previous digit, we do not need to compute them again and can skip Steps 1 and 2 in the computation of f_1. This leaves us with the following costs:
  - Field additions: $p^{2k} - p^k - 1$
  - Field multiplications: $2p^k - 3$

- Constant multiplications: 0
- Multiplicative depth: $\lceil \log_2(p^k - 1) \rceil + \ell$

To summarize (with $\texttt{eff}()$ denoting above costs), we have the following effort:

Case 1: $0 \leq x \leq p^k - 1$**.** This means that our number x can be encoded with 1 digit, and the result will have two digits. Remembering that we do not need to compute Steps 1 and 2 when computing f_1, the effort is $\texttt{eff}(c_0) + \texttt{eff}(r_1) = \texttt{eff}(c_0) + \texttt{eff}(f_1)$, so we have:

	Additions	**Field Multiplications**	**Constant Multiplications**	**Depth**
c_0	$p^{2k} + 2p^k - 3$	$2p^k \cdot \log_2(p^k) + p^k - 4$	p^{2k}	$\lceil \log_2(p^k - 1) \rceil + 1$
c_1	$p^{2k} - 2p^k$	$p^k - 1$	0	$\lceil \log_2(p^k - 1) \rceil + 1$
Total	$2p^{2k} - 3$	$2p^k \cdot \log_2(p^k) + 2p^k - 5$	p^{2k}	$\lceil \log_2(p^k - 1) \rceil + 1$

Case 2: $p^k \leq x$**.** This means that x will be encoded with $2 \leq \ell := \lfloor \log_p(x) \rfloor + 1$ digits and the result will have $\ell + 1$ digits. The effort is $\texttt{eff}(c_0) + \texttt{eff}(c_1) + (\ell - 2) \cdot \texttt{eff}(c_i) + \texttt{eff}(c_\ell)$, so we get:

	Additions	**Field Multiplications**	**Constant Multiplications**	**Depth**
c_0	$p^{2k} + 2p^k - 3$	$2p^k \cdot \log_2(p^k) + p^k - 4$	p^{2k}	$\lceil \log_2(p^k - 1) \rceil + 1$
c_1	$3p^{2k} - 2$	$2p^k \cdot \log_2(p^k) + 3p^k - 4$	$2p^{2k}$	$\lceil \log_2(p^k - 1) \rceil + 2$
c_i	$3p^{2k} + p^k - 2$	$2p^k \cdot \log_2(p^k) + 4p^k - 5$	$2p^{2k}$	$\lceil \log_2(p^k - 1) \rceil + i$
c_ℓ	$p^{2k} - p^k - 1$	$2p^k - 3$	0	$\lceil \log_2(p^k - 1) \rceil + \ell$
Total	$(3\ell - 1) \cdot p^{2k}$ $+(\ell - 1) \cdot p^k$ $-2\ell - 2$	$2\ell \cdot p^k \cdot \log_2(p^k)$ $+(4\ell - 2) \cdot p^k$ $-5\ell - 1$	$(\ell + 1) \cdot p^{2k}$	$\lceil \log_2(p^k - 1) \rceil + \ell$

We now compare the calculated effort to:

1. Encoding the number in base p instead of p^k and performing the addition.
2. Encoding the number in base p' with p' close to p^k.

Figure 2.5 shows the effort of adding two numbers of same size ($x = 20/7000/10^7$) in p^k-adic encoding for p^k up to 1000. Blue crosses are $\mathbb{Z}_p$, pink circles p^2, yellow triangles p^3, and the black square groups all bases p^k with $k \geq 4$, since the primes p with $p^k \leq 1000$ for increasing k become very few. We have omitted a graph for constant multiplications because there are none (when the scheme supports subtraction) when the plaintext space is $\mathbb{Z}_p$.

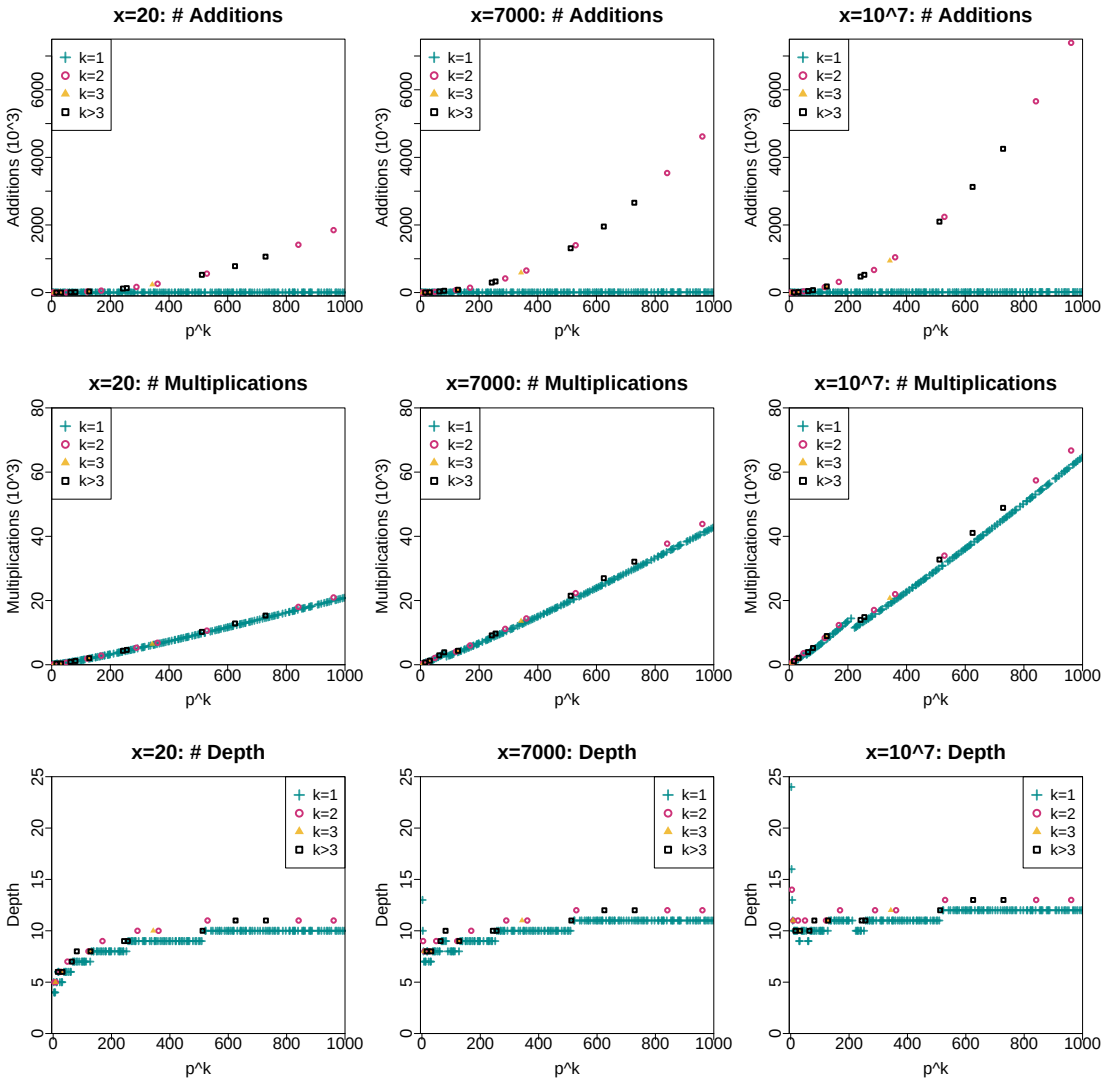

Figure 2.5: Number of field additions, field multiplications and multiplicative depth for multiplying $x = 20$ (first column)/$x = 7000$ (second column)/$x = 10^7$ (right column) to a number of same size for encoding base $GF(p^k)$.

We see that the p^k-encoding performs poorly regarding all metrics, and using $\mathbb{Z}_p$ as an encoding base is the better choice. For multiplication, the difference is not that obvious due to our optimization of calculating the l_i-functions only once and using them in different functions. Without this optimization (which does increase the required memory), multiplication for $k > 1$ performs significantly worse than for $k = 1$. We have manually checked (as it is hard to see for small values of p^k in Figure 2.5) that even with the optimization, the effort for $k > 1$ is indeed always bigger than for the next prime (i.e., the next choice for $k = 1$). For addition (and thus also for the total number of operations), the effort for $k > 1$ is enormously bigger than for $k = 1$, and the depth is always lower when choosing $k = 1$.

2.6 CONCLUSION

In conclusion, we showed in Section 2.4 that the smaller the encoding base p for a plaintext space of $\mathbb{Z}_p$, the smaller the cost in terms of ciphertext additions and multiplications, and that the optimal base in terms of multiplicative depth varies. However, the factor that induces this variation is the required encoding length of the input. We have seen in Section 2.5 that choosing the encoding base as $GF(p^k)$ with $k > 1$ always performs worse than choosing the encoding base as $\mathbb{Z}_p$. Since we can choose a prime p' that is close to any p^k (and thus requires roughly the same encoding length) which requires much less effort as shown in Figure 2.5, there is no case where choosing $GF(p^k)$ as an encoding base with $k > 1$ brings any benefit. For this reason, p^k is always the worst encoding choice and we are never without an alternative, so we do not continue with the analysis of $GF(p^k)$ as an encoding base.
We have argued in Section 1.3.2 that with digit-wise encoding, bootstrapping quickly becomes unavoidable and thus depth is not as important as the number of multiplications or total operations. Also, the optimum for depth depends heavily on the number of digits of the input and on the specific function, so that choosing the depth-optimal encoding base may even leak unwanted information about the function. For this reason, we will focus on $p = 2$ (and $k = 1$) in the remainder of this work, as this is optimal in terms of everything except depth.

Chapter 3

INCORPORATING NEGATIVE NUMBERS

In this chapter, we will extend our analysis to include negative numbers, focusing on the base $\mathbb{Z}_2$ as determined in the previous chapter. To this end, we first present in **Section 3.1** some elementary functions for binary encoding, which we will utilize as building blocks in subsequent sections. We then examine the two most popular encodings: Two's Complement (**Section 3.2**), which is most commonly used, and Sign-Magnitude (**Section 3.3**), which captures the straightforward idea of merely adding a sign bit to a natural number in binary encoding. Generally, accommodating for signed numbers will need one bit more than the corresponding unsigned number. Comparing the two encodings side by side in **Section 3.4**, we will see that Two's Complement performs better when adding two numbers, but Sign-Magnitude is superior for multiplying them. Thus, in **Section 3.5**, we present a mix of the two, called Hybrid Encoding, which switches between the two encodings to always utilize the better one for the operation at hand. We will see in Chapter 5 the impact this Hybrid Encoding has in practice. To aid the reader at later points, all the algorithms from this chapter and their respective costs are also presented as a summary in Appendix A.
This chapter is largely taken from [JA16].

3.1 ELEMENTARY FUNCTIONS

At this point, we briefly present some elementary functions on binary numbers that are useful in computing on encrypted data and will be utilized as building blocks in our computations on signed integers in the rest of this chapter.

3.1.1 Multiplexing

A multiplexer (MUX) gate is basically a shorthand way of expressing an `IF-ELSE` instruction. Concretely, let $c \in \{0, 1\}$ be the conditional, and let a be the value that is assigned if $c = 1$, and b the value that is used if $c = 0$. Then

$$\texttt{MUX}(c, a, b) := \begin{cases} a, & c = 1 \\ b, & c = 0. \end{cases}$$

The library we use later on indeed has a MUX gate available, but any FHE scheme over $\{0,1\}$ can implement this functionality through bit additions and multiplication:

$$\texttt{MUX}(c,a,b) = c \cdot a + (1+c) \cdot b = c \cdot (a+b) + b.$$

Thus, we count the effort of one MUX operation as:

	Field Additions	**Field Multiplications**	**Depth**
Total	2	1	$\max\{\texttt{depth}(a), \texttt{depth}(b), \texttt{depth}(c)\} + 1$

By abuse of notation, we may also write $\texttt{MUX}(c,a,b)$ for $c \in \{0,1\}$ and a and b bitstrings – in this case, the `MUX` gate is applied to each bit, and the above addition and multiplication costs are multiplied by the bitlength. The depth propagation remains the same in that the maximum input depth increases by 1.

3.1.2 Comparison of Unsigned Numbers

We now show how to compare two natural numbers (i.e., check whether $a < b$ for two natural numbers a and b) in Algorithm 1, and then later use that as a building block to compare signed numbers.

Algorithm 1: `Unsigned Comparison`

Input: Natural number $a = a_n \dots a_1 a_0$
Input: Natural number $b = b_n \dots b_1 b_0$
// Set result to 0
1 $res = 0$
2 **for** $i = 0$ *to* n **do**
  // Set temp to 0 if $a_i \neq b_i$ and to 1 if $a_i = b_i$
3 $temp = a_i + b_i + 1$
  // If $temp = 1$ (inputs are equal), don't change res. If $temp = 0$ (inputs are unequal), set $res = b_i$
4 $res = \texttt{MUX}(temp, res, b_i)$
5 **end**
// $res = 1 \Leftrightarrow a < b$
Output: res

The idea[1] is that the variable res is set to 0 and then in each iteration of the for-loop it denotes the result of the comparison on the previous bits. Thus, if the two bits at position i are equal ($a_i = b_i$), the result res of the comparison does not change. If they are unequal, the lower bits do not matter anymore and the result is set to $res = b_i$. This works because if $b_i = 1$, that means $a_i = 0$, so the number $a_i \dots a_1 a_0$ is smaller than $b_i \dots b_1 b_0$. Thus the outcome should be $1 = b_i$. If, on the other hand, $b_i = 0$, then a_i=1 and the number $a_i \dots a_1 a_0$ is larger than $b_i \dots b_1 b_0$, so the outcome is $0 = b_i$.
Of course, starting from the LSB is a bit counter-intuitive: In unencrypted computations, we would start from the MSB because we could terminate at the first position where the

[1] Credited to the TFHE team `https://tfhe.github.io/tfhe/tuto-cloud.html` for this elegant notation, last accessed May 29, 2018.

bits differ. However, in encrypted computations we cannot see when this is the case and would have to iterate over the entire length of the inputs anyway instead of stopping early, so the above method, which uses less elementary operations, is the more efficient way of computing the comparison.
Note that by changing Line 1 to $res = 1$, we can test for $a \leq b$ instead of $a < b$ with exactly the same effort.
With this algorithm, the effort of comparing two n-bit numbers is:

Line	**Field Additions**	**Field Multiplications**	**Depth**
3	$2n$	0	0
4	$2n$	n	n
Total	$4n$	n	n

3.1.3 Addition of Unsigned Numbers

Recall from Remark 1 in Section 2.3.3 that for $p = 2$, there is an even more efficient way of writing the formula to compute the carry (because $x^2 = x \bmod 2$):

$$r_i = (a_{i-1} + b_{i-1}) \cdot (a_{i-1} + r_{i-1}) + a_{i-1}.$$

This is the majority function, as the carry will be 1 if at least two of the three values $a_{i-1}, b_{i-1}, r_{i-1}$ are 1.
Thus, we quickly examine the effort for adding two unsigned numbers in binary encoding.

Additions: Using the above formula for the carry, we again have $c_0 = a_0 + b_0$ (because $r_0 = 0$) and then $c_i = a_i + b_i + r_i$ for $i = 1 \ldots n-1$. For the last bit $c_n = r_n$, we only have the additions from r_n. From the computation of r_i, we get 0 additions for $r_0 = 0$ and $r_1 = a_0 \cdot b_0$, and 3 additions for $i = 2, \ldots, n$.

Multiplications: Likewise, we get 0 multiplications from $c_i = a_i + b_i + r_i$ and $r_0 = 0$, 1 multiplication from $r_1 = a_0 \cdot b_0$, and 1 multiplication from the computation of r_i for $i = 2, \ldots, n$.

Depth: Lastly, for the multiplicative depth, we see that r_1 has depth 1, and then depth increases by 1 for $i = 2, \ldots, n$ because r_i is multiplied with another value $(a_{i-1} + b_{i-1})$. We obtain a total depth of n.

Thus, in total, to add two n-bit numbers, we get an effort of:

	Additions	**Multiplications**	**Depth**
c_0	1	0	0
c_1	2	1	1
c_i	5	1	$+1$ $(= i)$
c_n	3	1	n
Total	$1 + 2 + 5 \cdot (n-2) + 3$ $= 5n - 4$	$0 + 1 + (n-2) + 1$ $= n$	n

3.1.4 Subtraction of Unsigned Numbers

We now explore how to subtract two unsigned numbers. Of course, the result must also be an unsigned number, so we require it to be non-negative – that is, we want to compute $a-b$ for two n-bit numbers a and b with $b \leq a$. We use a procedure called the *Complement Method:*

1. Flip all bits of $b = b_{n-1} \dots b_1 b_0$ to obtain $\bar{b} = \bar{b}_{n-1} \dots \bar{b}_1 \bar{b}_0$.
2. Compute $a - b = a + \bar{b} + 1$ using unsigned addition and discarding the $(n+1)^{\text{th}}$ result bit.

Lemma 7. *The Complement Method is correct.*

Proof: Consider that $\bar{b}_i = 1 - b_i$. Then we can write (over $\mathbb{N}$):

$$\begin{aligned} a + \bar{b} + 1 &= a + \left(\sum_{i=0}^{n-1} \bar{b}_i \cdot 2^i \right) + 1 \\ &= a + \left(\sum_{i=0}^{n-1} (1 - b_i) \cdot 2^i \right) + 1 \\ &= a + \left(\sum_{i=0}^{n-1} 2^i \right) - \left(\sum_{i=0}^{n-1} b_i \cdot 2^i \right) + 1 \\ &= a + (2^n - 1) - b + 1 \\ &= 2^n + a - b. \end{aligned}$$

Since we discard the $(n+1)^{\text{th}}$ result bit, the term 2^n disappears from the sum, leaving us with just $a - b$ as promised.

□

We see that we have an effort of n bit additions to flip the bits in Step 1, and then the effort of two unsigned additions of n-bit numbers, but without the effort for the $(n+1)^{\text{th}}$ result bit. Thus, we get:

Step	**Additions**	**Multiplications**	**Depth**
1	n	0	0
2	$2 \cdot (5n-7)$	$2 \cdot (n-1)$	$n-1$
Total	$11n-14$	$2n-2$	$n-1$

Regarding depth, we have the same reasoning we had in Section 2.4.2.2: After adding $a+\overline{b} := d$, digit d_i of the result has depth i. Inputting this into the second addition with 1 (where each bit has depth 0), the carry adds a depth increase of 1 with each bit. However, this is also the depth of the next input bit: $c_i = \alpha + d_i + r_i$ where $\alpha \in \{0,1\}$ with depth 0 and d_i has the same depth as r_i and thus c_i. Thus, the total maximum depth does not increase through the second addition.

3.1.5 Multiplication of Unsigned Numbers

As we already stated in Section 2.4.2, the case $p = 2$, which we are working with for the rest of this thesis, is a special case because we have no carry in the computation of the rows of the multiplication matrix (recall Definition 2.2). Since this results in a better performance than the previous chapter suggests, we present the cost of multiplying two unsigned binary numbers with encoding base $\mathbb{Z}_2$ at this point.
Suppose we are computing $a_{n-1}\dots a_1a_0 \cdot b_{m-1}\dots b_1b_0$. Then the multiplication matrix has m rows, each of length n and indented by $i-1$ positions. We again use the trick of copying the excess bits of the upper row from Section 2.4.2.2. We can see that we first do $\frac{m}{2}$ additions of length n (because only the lower row counts effort-wise), which result in $\frac{m}{2}$ rows of length $n+2$ (because the result is one longer, and we copied one bit at the back). If m is not divisible by 2, we keep the row and will perform another addition with effort n later – thus, we see that overall, we will perform $\left\lceil \frac{m}{2} \right\rceil$ additions of length n. Next, we do $\left\lceil \frac{m}{4} \right\rceil$ additions of length $n+2$, and the result is $n+5$ (because we copy 2 bits at the back, and the result is one bit longer). We do this until we are left with only 1 row – so $\lfloor \log_2(m) \rfloor$ times.
In total, with `NatAdd` denoting the addition of unsigned numbers from Section 3.1.3, we get an effort of:

$$\sum_{i=1}^{\lfloor \log_2(m) \rfloor} \left\lceil \frac{m}{2^i} \right\rceil \cdot \texttt{Eff}(\texttt{NatAdd}(n + 2^{i-1} + i - 2)),$$

where `Eff` is the number of additions or multiplications, respectively. Regarding depth, we again have the effect that the carry increases the depth by 1 from the maximum of its three inputs $(r_{i-1}, a_{i-1}, b_{i-1})$. After one addition, bit i of the output has depth i. However, the depth does not increase when inputting this into a further addition: As an example, the input bits a_3 and b_3 may have depth 3, but the carry from position 2 also has depth 3, so in adding $c_3 = a_3 + b_3 + r_3$, we get a depth of 3 as we would with fresh ciphertexts where each input bit has depth 0. Since we start off with depth 0, and then bit i has depth i in the addition, we get a maximum depth of $m+n-1$.

We now move on to incorporating signed numbers.

3.2 TWO'S COMPLEMENT

The most popular encoding that incorporates negative numbers is called *Two's Complement*: Here, we write an integer a as

$$a = a_n \cdot (-2^n) + \sum_{i=0}^{n-1} a_i \cdot 2^i$$

where $n = \lfloor \log_2(a) \rfloor + 1$ and $a_i \in \{0, 1\}$. This means that the most significant bit (MSB) encodes the negative value -2^n and is thus 1 exactly when $a < 0$.

Example 3.1: *Consider the bitstring* 1011*: This encodes*

$$1 \cdot (-2^3) + 0 \cdot 2^2 + 1 \cdot 2 + 1 \cdot 1 = -8 + 2 + 1 = -5.$$

Conversely, to encode the number 12*, we write*

$$12 = 0 \cdot (-16) + 1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 1 = 0 \cdot (-2^4) + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2 + 0 \cdot 1,$$

so we encode the number as 01100*. To encode its negative* -12*, we write*

$$-12 = 1 \cdot (-16) + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 1 = 1 \cdot (-2^4) + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2 + 0 \cdot 1,$$

yielding an encoding of 10100*.*

One important concept when working with Two's Complement is that of increasing the bitlength of a number while keeping its original value.

Definition 3.1 (Sign Extension)**.** Let $a = a_n \dots a_1 a_0$ be an $n+1$-bit number in Two's Complement-Encoding. Then *sign extension* turns a into an $n+1+k$-bit number for any $k \in \mathbb{N}$ by prepending k copies of the most significant bit a_n to the bitstring:

$$a = a_n \dots a_1 a_0 \mapsto \underbrace{a_n \dots a_n}_{k \text{ times}} a_n \dots a_1 a_0.$$

This operation does not change the value of a:

Lemma 8. *Sign extension does not change the value of the underlying number.*

Proof: Let $a = a_n \dots a_1 a_0$ be the original number and

$$\underbrace{a_n \dots a_n}_{k \text{ times}} a_n \dots a_1 a_0$$

its sign extension by k bits. Then originally, we have

$$a = a_n \cdot (-2^n) + \sum_{i=0}^{n-1} a_i \cdot 2^i.$$

The sign extended bitstring has the value

$$\begin{aligned}
a' &= a_n \cdot (-2^{n+k}) + \sum_{i=0}^{n+k-1} a_i \cdot 2^i \\
&= a_n \cdot (-2^{n+k}) + \sum_{i=n}^{n+k-1} a_n \cdot 2^i + \sum_{i=0}^{n-1} a_i \cdot 2^i \\
&= a_n \cdot (-2^{n+k}) + \sum_{i=n}^{n+k-1} a_n \cdot 2^i + \sum_{i=0}^{n-1} a_n \cdot 2^i - \sum_{i=0}^{n-1} a_n \cdot 2^i + \sum_{i=0}^{n-1} a_i \cdot 2^i \\
&= a_n \cdot (-2^{n+k}) + a_n \cdot \sum_{i=0}^{n+k-1} 2^i - a_n \cdot \sum_{i=0}^{n-1} 2^i + \sum_{i=0}^{n-1} a_i \cdot 2^i \\
&= a_n \cdot (-2^{n+k}) + a_n \cdot (2^{n+k} - 1) - a_n \cdot (2^n - 1) + \sum_{i=0}^{n-1} a_i \cdot 2^i \\
&= -a_n \cdot 2^{n+k} + a_n \cdot 2^{n+k} - a_n - a_n \cdot 2^n + a_n + \sum_{i=0}^{n-1} a_i \cdot 2^i \\
&= a_n \cdot (-2^n) + \sum_{i=0}^{n-1} a_i \cdot 2^i,
\end{aligned}$$

which is exactly the original value of a.

□

The most important operations for numbers encoded in this fashion and the effort incurred for computing them are presented in the following.

3.2.1 Addition in Two's Complement

Addition in Two's Complement encoding works in much the same way as unsigned binary addition, except for one point: To obtain the correct result when adding two n-bit numbers, the result must also be encodeable by n bits, and any values past the n^{th} bit are discarded. Since the result of adding two n-bit numbers is usually $n + 1$ bits long, we first perform sign extension by one bit on the inputs so that we can then add two $n + 1$ bit numbers whose sum is also encodeable by $n + 1$ bits, thus yielding the correct result.

Example 3.2: *Suppose we are adding the* 4*-bit numbers*

$$-5 = 1011 \quad \textit{and} \quad 7 = 0111 :$$

First, we use sign extension to obtain the 5*-bit numbers*

$$-5 = 11011 \quad \textit{and} \quad 7 = 00111,$$

then we carry out normal addition, resulting in the number

$$11011 + 00111 = 100010.$$

Discarding the excess leftmost bit, we obtain 00010*, which indeed decodes to* $2 = -5 + 7$*.*

Note at this point that addition has been defined for two numbers of equal length, but if this is not the case we can easily increase the bitlength of the shorter one through sign extension as described above so that the lengths match.

3.2.1.1 *Effort*

Recalling the effort of unsigned addition from Section 3.1.3, we realize that to add two n-bit numbers in Two's Complement encoding, we have the same effort as for adding two $n+1$-bit unsigned binary numbers, but without the effort of the last bit c_{n+1} we would have there.
In total, to add two n-bit numbers in Two's Complement encoding, we get an effort of:

	Additions	**Multiplications**	**Depth**
c_0	1	0	0
c_1	2	1	1
c_i	5	1	$+1$ $(= i)$
Total	$1 + 2 + 5 \cdot (n-1)$ $= 5n - 2$	$0 + 1 + (n-1)$ $= n$	$0 + 1 + (n-1)$ $= n$

3.2.2 Multiplication in Two's Complement

When multiplying two numbers in Two's Complement encoding, we need to follow a few steps. For maximum generality, we assume that our numbers have lengths m and n, respectively.

1. Increase the bitlength of both numbers through sign extension (as described above) to length $m + n$.

2. Perform regular binary multiplication of the two resulting numbers. Note that to add the individual rows, we must use the addition function from above.

3. Keep only the rightmost $n + m$ bits.

Example 3.3: *As an example, we multiply* $-3 = 101$ *(*$m = 3$*) and* $7 = 0111$ *(*$n = 4$*): First, we sign extend both numbers to length* $n+m = 4+3 = 7$ *to obtain* $-3 = 1111101$ *and* $7 = 0000111$*. We then compute (always only performing addition up to the cutoff-position indicated by the vertical dashed line):*

$$
\begin{array}{rrrrrrrcrrrrrrr}
1 & 1 & 1 & 1 & 1 & 0 & 1 & \cdot & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
\hline
 & & & & & & & & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
 & & & & & & & & 1 & 1 & 1 & 1 & 0 & 1 & \\
 & & & & & & & & 1 & 1 & 1 & 0 & 1 & & \\
 & & & & & & & & 0 & 0 & 0 & 0 & & & \\
 & & & & & & & & 0 & 0 & 0 & & & & \\
 & & & & & & & & 0 & 0 & & & & & \\
 & & & & & & & & 0 & & & & & & \\
\hline\hline
 & & & & & & & & 1 & 1 & 0 & 1 & 0 & 1 & 1
\end{array}
$$

This correctly yields $1101011 = -21$.

Whenever we refer to the addition of rows during multiplication, we are referring to the rows of the multiplication matrix (see Definition 2.2) as in the above example, which has 7 rows of lengths $7, 6, \ldots, 1$, respectively.

3.2.2.1 *Removing Redundancy*

As can easily be seen, the above approach introduces redundancy because the same product is calculated several times, which may become costly if bit multiplication is an expensive operation. As a minimal example, consider the case for $n = m = 2$:

$$
\begin{array}{rrrrcrrrr}
a_1 & a_1 & a_1 & a_0 & \cdot & b_1 & b_1 & b_1 & b_0 \\
\hline
 & & & & & a_1 \cdot b_0 & a_1 \cdot b_0 & a_1 \cdot b_0 & a_0 \cdot b_0 \\
 & & & & & a_1 \cdot b_1 & a_1 \cdot b_1 & a_0 \cdot b_1 & \\
 & & & & & a_1 \cdot b_1 & a_0 \cdot b_1 & & \\
 & & & & & a_0 \cdot b_1 & & & \\
\hline\hline
\end{array}
$$

Here, we can see that out of the 10 terms that occur

$$\text{(generally: } (n+m) + (n+m-1) + \cdots + 2 + 1 = \frac{(n+m)\cdot(n+m+1)}{2} \text{ terms)},$$

the term $a_0 \cdot b_0$ occurs once, and the terms $a_1 \cdot b_0$, $a_0 \cdot b_1$ and $a_1 \cdot b_1$ all occur three times. As can easily be seen, we actually only have $n \cdot m$ different products, so we can save a significant amount of computation by avoiding this redundancy.
We now show in detail how to avoid computing the same product several times, which more than halves the effort of the matrix computation step by bringing it from $\frac{(n+m)\cdot(n+m+1)}{2}$ to $n \cdot m$. To this end, we will think of the rows that are generated as an $(n+m) \times (n+m)$

- matrix A (with some empty entries) indexed as $A[i, j]$, where i refers to the row and j to the bit position in the row, i.e., 0 is on the right-hand side.
As an example for the case with $n = m = 2$ (where we write $[i, j]$ instead of $A[i, j]$), we have the following multiplication matrix:

a_1	a_1	a_1	a_0	$\cdot$	b_1	b_1	b_1	b_0
					$[0,3]$	$[0,2]$	$[0,1]$	$[0,0]$
					$[1,3]$	$[1,2]$	$[1,1]$	
					$[2,3]$	$[2,2]$		
					$[3,3]$			

With this notation in place, we present the pseudocode instructions for reducing the number of computations in multiplying two numbers of lengths n and m in Algorithm 2.

Algorithm 2: `Redundancy Reduction`

Input: $a_{n-1} \dots a_1 a_0,\ b_{m-1} \dots b_1 b_0$
for $0 \leq j \leq m-1$ **do**
  for $0 \leq i \leq n-1$ **do**
    $A[j, j+i] = a_i \cdot b_j$
  end
  for $n+j \leq i \leq n+m-1$ **do**
    $A[j, i] = A[j, n+j-1]$
  end
end
for $m \leq i \leq n+m-1$ **do**
  for $i \leq j \leq n+m-1$ **do**
    $A[i, j] = A[i-1, j-1]$
  end
end
Output: A

The output A is now the same multiplication matrix (see Definition 2.2) as we would obtain by straightforward computation, but we save some effort by computing it this way because we copy duplicate values rather than computing them anew for each position. The second step of summing up the rows of this matrix is then done as before.
Note that in the simpler case where one value is known, i.e., multiplication by a constant, we do not need to do quite as much work: For simplicity, we always assume that the input b is known. We again first need to do sign extension for Two's Complement, but in the next step instead of having to compute $n \cdot m$ terms $a_i \cdot b_j$ as before, we can just copy the string a for every bit that is 1 in b, shifting to the left with each bit. This way, we save $n \cdot m$ multiplications from the generation of the matrix and reduce the depth by one. Also, note that we now don't need to add as many rows, as we only write down those that correspond to the non-zero bits in b. Thus, we only need to do $\mathtt{hm}(b)$ row additions, where $\mathtt{hm}(b)$ is the hamming weight of b. Of course, the complexity and multiplicative depth now depend on the value of b and are the same as for regular multiplication in the worst case. However, on average we will only have to do half as many row additions.

3.2.2.2 *Effort*

We now compute the effort of multiplying two numbers in Two's Complement encoding. We use the above method to avoid redundancy, and also copy the bits above blank spaces to the result rather than padding the shorter row as presented in Section 2.4.2.2, which means we only have the effort of the shorter row length in the addition. As before, we add row 1 to row 2, then add that to the result of adding row 3 to row 4 etc., until we are left with the final result. Note that we disregard (i.e., do not compute) anything past the $(m+n)^{\text{th}}$ bit.

Example 3.4: *Suppose we have generated the rows of our matrix as*

$$\begin{array}{ccccc|ccccc} a_1 & a_1 & a_1 & a_1 & a_0 & \cdot & b_2 & b_2 & b_2 & b_1 & b_0 \\ \hline & & & & & & c_4 & c_3 & c_2 & c_1 & c_0 \\ & & & & & & d_3 & d_2 & d_1 & d_0 & \\ & & & & & & e_2 & e_1 & e_0 & & \\ & & & & & & f_1 & f_0 & & & \\ & & & & & & g_0 & & & & \end{array}$$

with the appropriate values in the matrix as above. Then we perform the following steps:

- ***Row 1+Row 2:***
$$\begin{array}{cccccc} & c_4 & c_3 & c_2 & c_1 & c_0 \\ + & d_3 & d_2 & d_1 & d_0 & \\ \hline & h_4 & h_3 & h_2 & h_1 & c_0 \end{array}$$
*The result is $h_4h_3h_2h_1c_0$, where $h_4h_3h_2h_1$ is the result of adding $c_4c_3c_2c_1$ and $d_3d_2d_1d_0$. Thus, we have the effort of one addition of **length 4**, and the result, denoted Row$_2$ 1, has length* 5*.*

- ***Row 3+Row 4:***
$$\begin{array}{cccc} & e_2 & e_1 & e_0 \\ + & f_1 & f_0 & \\ \hline & i_2 & i_1 & e_0 \end{array}$$
*The result is $i_2i_1e_0$, where i_2i_1 is the result of adding e_2e_1 and f_1f_0. Thus, we have the effort of one addition of **length 2**, and the result, denoted Row$_2$ 2, has length* 3*.*

- ***Row$_2$ 1+Row$_2$ 2:***
$$\begin{array}{ccccc} & h_4 & h_3 & h_2 & h_1 & c_0 \\ + & i_2 & i_1 & e_0 & & \\ \hline & j_4 & j_3 & j_2 & h_1 & c_0 \end{array}$$
*The result is $j_4j_3j_2h_1c_0$, where $j_4j_3j_2$ is the result of adding $h_4h_3h_2$ and $i_2i_1e_0$. Thus, we have the effort of one addition of **length 3**, and the result, denoted Row$_3$ 1, has length* 5*.*

- ***Row$_3$ 1+Row 5:***
$$\begin{array}{ccccc} & j_4 & j_3 & j_2 & h_1 & c_0 \\ + & g_0 & & & & \\ \hline & k_4 & j_3 & j_2 & h_1 & c_0 \end{array}$$
*The result is $k_4j_3j_2h_1c_0$, where k_4 is the result of adding j_4 and g_0. Thus, we have the effort of one addition of **length 1**, and the result has length* 5*.*

It is easy to see that for a length of $n+m$, we always have one addition each of length $1, 2, \ldots, n+m-1$. The reason is that the addition effort is always the length of the smaller of the two inputs, and the result has the length of the larger of the two, which thus gets input as the shorter input at some later point unless it is the final length $n+m$.
A row addition of length k has the same effort as a Two's Complement addition of length $k-1$ (recall the table in Section 3.2.1), as the sign extension has already been done in the matrix. Thus, for length 1, we only compute $a_0 + b_0$ (1 addition), and for length 2, we compute c_0 and c_1 for a total cost of 3 additions, 1 multiplication and a depth of 1. For all lengths ℓ larger than 2, we have a cost of $5(\ell-1)-2 = 5\ell-7$ additions, $\ell-1$ multiplications, and a depth of $\ell-1$. Note that to compute the rows of the matrix, we need $m \cdot n$ multiplications and a depth of 1.
Regarding depth, note that the matrix entries start with a depth of 1, and then the depth propagates through the addition. We always have $\texttt{depth}(c_0) = \texttt{depth}(c_1) = 1$ in each row because c_0 is copied from the longer row and $c_1 = a_1 + b_1$. The first carry $r_2 = a_1 \cdot b_1$ has depth 2, which is thus also the depth of c_2. (Since the result bit in the addition is $c_i = a_i + b_i + r_i$, we see that its depth is the maximum of the depth of the three input bits.) As we continue, the carry r_i increases by 1 with each bit to the left and the inputs a_i and b_i also have depth at most i by the exact same reasoning for the addition that yielded the respective input row. Thus, the result bit c_i has depth i, and since our longest row is $m+n$ bits long, the maximum depth we get is $m+n-1$.

In total, to multiply two numbers of lengths m and n in Two's Complement encoding, we have an effort of:

- **Additions:** $1 + 3 + \sum_{i=3}^{n+m-1} 5i - 7 = \frac{5(m^2+n^2)-19(m+n)}{2} + 5mn + 10$
- **Multiplications:** $m \cdot n + 1 + \sum_{i=3}^{n+m-1} (i-1) = m \cdot n + \sum_{i=1}^{n+m-2} i = m \cdot n + \frac{(n+m-2)\cdot(n+m-1)}{2}$
- **Depth:** $m+n-1$

3.2.3 Negation in Two's Complement

Negating a given number (i.e., $a \mapsto -a$) in Two's Complement works as follows:

1. Flip all bits (i.e., XOR them with 1).
2. Add 1 to the resulting number using unsigned addition and discarding the $(n+1)^{\text{th}}$ result bit.

Example 3.5: *Take the number*

$$-3 = 101:$$

Flipping all bits gives us 010*, and adding* 1 *yields*

$$011 = 3.$$

Correctness can be seen as in Section 3.1.4.

3.2.3.1 *Effort*

We see that to invert an n-bit number we need to perform n field additions for the first step, and then incur the effort of one $(n-1)$-bit Two's Complement addition. This gives us a total effort of:

Step	**Additions**	**Multiplications**	**Depth**
1	n	0	0
2	$5(n-1)-2$	$n-1$	$n-1$
Total	$6n-7$	$n-1$	$n-1$

3.2.4 Comparison in Two's Complement

Recall that in Section 3.1.2, we presented the comparison of unsigned binary numbers in Algorithm 1. Denoting this subroutine as `NatComp`, we can compare two numbers in Two's Complement encoding as shown in Algorithm 3.

Algorithm 3: `Two's Complement Comparison`(a, b)

Input: Signed number $a = a_n \dots a_1 a_0$
Input: Signed number $b = b_n \dots b_1 b_0$
`// Compare as if natural numbers, result is correct if signs are equal`
1 $c = \texttt{NatComp}(a, b)$
`// The sign bits are` a_n `and` b_n
2 $res = a_n \cdot (b_n + 1) + (a_n + b_n + 1) \cdot c$
Output: res

As can easily be verified, the formula in Line 2 evaluates to c if $a_n = b_n$, i.e., the signs are equal – in this case, the result of $\texttt{NatComp}(a, b)$ is correct. If $a_n = 0$ and $b_n = 1$, i.e., b is negative and a is positive, the formula evaluates to 0, which is correct because $a \not< b$. Lastly, if $a_n = 1$ and $b_n = 0$, i.e., b is positive and a is negative, the formula evaluates to 1, which is also correct because $a < b$. The values can also be seen in the following table:

a_n	b_n	$a_n \cdot (b_n + 1) + (a_n + b_n + 1) \cdot c$
0	0	c
0	1	0
1	0	1
1	1	c

Switching from $a < b$ to $a \leq b$ can be achieved by simply changing the subroutine `NatComp` appropriately as described in Section 3.1.2.

3.2.4.1 *Effort*

To compare two n-bit numbers in Two's Complement encoding, we thus have an effort of:

Line	Field Additions	Field Multiplications	Depth
1	$4n$	n	n
2	4	2	$+1$
Total	$4n+4$	$n+2$	$n+1$

3.3 SIGN-MAGNITUDE

In contrast to Two's Complement encoding, Sign-Magnitude formalizes the most intuitive idea of using regular binary encoding and having an extra bit that determines the sign. Conventionally, this is the most significant bit, which is 1 when a number is negative and 0 when a number is positive. Concretely, elements in this encoding have the form $a_n a_{n-1} \dots a_1 a_0$ with $a_i \in \{0,1\}$, where

$$a = (-1)^{a_n} \cdot \sum_{i=0}^{n-1} a_i \cdot 2^i.$$

It is easy to see that for positive numbers, this encoding is the same as Two's Complement. This encoding suffers from having two representations of 0: $00\dots0$ and $10\dots0$.

Example 3.6: *Consider the bitstring* 1101*: This encodes*

$$(-1)^1 \cdot \left(1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0\right) = (-1) \cdot (4+1) = -5.$$

Conversely, its inverse 5 *is encoded as*

$$1 \cdot (4+1) = (-1)^0 \cdot \left(1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0\right),$$

which is expressed as 0101*.* ———

Note that to get the absolute value of a number in this encoding (which we will sometimes need to do in the following), it suffices to delete the MSB and treat the number as an unsigned integer.

3.3.1 Addition in Sign-Magnitude

Adding two numbers in Sign-Magnitude encoding is surprisingly complex, as we must consider different cases and combine them via multiplexing. As subroutines, we will need the following components from Section 3.1:

- Unsigned Addition $\texttt{NatAdd}(a,b) = a+b$ for $0 \le a, b$
- Unsigned Subtraction: $\texttt{NatSub}(a,b) = a-b$ if $0 \le b \le a$
- Unsigned Comparison: $\texttt{NatComp}(a,b) = 1 \Leftrightarrow a < b$.

Algorithm 4: `Sign-Magnitude Addition`

Input: Signed number $a = a_n \dots a_1 a_0$
Input: Signed number $b = b_n \dots b_1 b_0$
`// Get numbers without sign bits, i.e.,` $|a|$ `and` $|b|$`.`
1 $\tilde{a} = a_{n-1} \dots a_1 a_0$
2 $\tilde{b} = b_{n-1} \dots b_1 b_0$
`//` res_1 `is the unsigned addition of` $|a|$ `and` $|b|$ `with` a`'s sign bit.`
3 $res_1 = a_n || \texttt{NatAdd}(\tilde{a}, \tilde{b})$
`//` res_2 `is the unsigned subtraction of` $|b| - |a|$ `with` b`'s sign bit.`
4 $res_2 = b_n || \texttt{NatSub}(\tilde{b}, \tilde{a})$
`//` res_3 `is the unsigned subtraction of` $|a| - |b|$ `with` a`'s sign bit.`
5 $res_3 = a_n || \texttt{NatSub}(\tilde{a}, \tilde{b})$
`//` $c_1 = 1$ `if` $a_n \neq b_n$`, and 0 if the signs are equal.`
6 $c_1 = a_n + b_n$
`//` $c_2 = 1$ `if` $\tilde{a} < \tilde{b}$`, and 0 otherwise.`
7 $c_2 = \texttt{NatComp}(\tilde{a}, \tilde{b})$
`//` $temp = res_2$ `if` $c_2 = 1$`, and` $temp = res_3$ `otherwise.`
8 $temp = \texttt{MUX}(c_2, res_2, res_3)$
`//` $res = temp$ `if` $c_1 = 1$`, and` $res = res_1$ `otherwise.`
9 $res = \texttt{MUX}(c_1, temp, res_1)$
Output: res

With these subroutines, we can express the addition of two numbers in Sign-Magnitude encoding as presented in Algorithm 4.
To see correctness, we will examine the workings of this algorithm. First, consider the conditional c_1 in Line 6: It is 1 if $a_n \neq b_n$ and 0 otherwise. The multiplexer in Line 9 assigns the output of the computation based on this conditional: If the conditional is 0 (i.e., a and b have the same sign), the result will be res_1, which is just the unsigned addition $|a| + |b|$ with the sign of a (and thus also b) concatenated.
If the conditional is 1, i.e., a and b have different signs, the result will be the value of $temp$ from Line 8. This itself is the output of a multiplexer on the conditional c_2: If $c_2 = 1$ (i.e., $|a| < |b|$), then $temp = res_2 = b_n || (|b| - |a|)$. This is correct because when adding two numbers of different signs, the result sign will be the sign of the input with the bigger absolute value, and the result absolute value will be the difference between the two input absolute values. Similarly, if $c_2 = 0$ (i.e., $|b| \leq |a|$), we get $temp = res_3 = a_n || (|a| - |b|)$.

3.3.1.1 *Effort*

To add two n-bit numbers in Sign-Magnitude encoding, we have an effort of:

Line	Field Additions	Field Multiplications	Depth
3	$5n-9$	$n-1$	$n-1$
4	$11n-25$	$2n-4$	$n-2$
5	$11n-25$	$2n-4$	$n-2$
6	1	0	0
7	$4n-4$	$n-1$	$n-1$
8	$2n$	n	$+1(=n)$
9	$2n+2$	$n+1$	$+1(=n+1)$
Total	$35n-60$	$8n-9$	$n+1$

3.3.2 Multiplication in Sign-Magnitude

In contrast to addition, multiplying two numbers in Sign-Magnitude encoding is conceptually simple: We simply delete the sign bits a_n and b_n, multiply the results as unsigned integers, and append the sign bit $a_n + b_n$.
Computing the sign of the result takes one Addition, and using the effort for unsigned multiplication from Section 3.1.5 (with input lengths $m-1$ and $n-1$, and `NatAdd` denoting the unsigned Addition from Section 3.1.3), we get an effort of:

- **Additions:** $1 + \sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \left\lceil \frac{m-1}{2^i} \right\rceil \cdot \texttt{Adds}(\texttt{NatAdd}(n + 2^{i-1} + i - 3))$
- **Multiplications:** $\sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \left\lceil \frac{m-1}{2^i} \right\rceil \cdot \texttt{Mults}(\texttt{NatAdd}(n + 2^{i-1} + i - 3))$
- **Depth:** $m + n - 3$.

Recalling that the addition of two n-bit numbers has $5n-4$ bit additions and n bit multiplications, we can also write this as:

- **Additions:** $1 + \sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \left\lceil \frac{m-1}{2^i} \right\rceil \cdot \left(5 \cdot (n + 2^{i-1} + i) - 19\right)$
- **Multiplications:** $\sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \left\lceil \frac{m-1}{2^i} \right\rceil \cdot \left(n + 2^{i-1} + i - 3\right)$
- **Depth:** $m + n - 3$.

3.3.3 Negation in Sign-Magnitude

Negation is very easy, as it can be obtained through a single bit addition: Since the MSB determines the sign, flipping it (i.e., setting $a_n = a_n + 1$) will transform a number a into its negative, $-a$.

3.3.3.1 *Effort*

To invert an n-bit number in Sign-Magnitude encoding, we thus have an effort of:

	Field Additions	**Field Multiplications**	**Depth**
Total	1	0	0

3.3.4 Comparison in Sign-Magnitude

Again denoting the comparison of unsigned binary numbers from Algorithm 1 in Section 3.1.2 as `NatComp`, we can compare two numbers in Sign-Magnitude encoding as shown in Algorithm 5.

Algorithm 5: `Sign-Magnitude Comparison`

Input: Signed number $a = a_n \dots a_1 a_0$
Input: Signed number $b = b_n \dots b_1 b_0$
`// Get numbers without sign bits, i.e.,` $|a|$ `and` $|b|$`.`
1 $\tilde{a} = a_{n-1} \dots a_1 a_0$
2 $\tilde{b} = b_{n-1} \dots b_1 b_0$
`// Compare absolute values as natural numbers`
3 $c =$ `NatComp`$(\tilde{a}, \tilde{b})$
`// The sign bits are` a_n `and` b_n
4 $res = a_n + (a_n + b_n + 1) \cdot c$
Output: res

The formula in Line 4 evaluates to c if $a_n = b_n = 0$, i.e., both numbers are positive – in this case, the result of `NatComp`$(\tilde{a}, \tilde{b})$ is correct. If $a_n = b_n = 1$, i.e., both numbers are negative, the formula evaluates to $1 + c$ – in this case, the result of `NatComp`$(\tilde{a}, \tilde{b})$ is wrong and is thus negated. If $a_n = 0$ and $b_n = 1$, i.e., b is negative and a is positive, the formula evaluates to 0, which is correct because $a \not< b$. Lastly, if $a_n = 1$ and $b_n = 0$, i.e., b is positive and a is negative, the formula evaluates to 1, which is also correct because $a < b$. The values can also be seen in the following table:

a_n	b_n	$a_n + (a_n + b_n + 1) \cdot c$
0	0	c
0	1	0
1	0	1
1	1	$c + 1$

The attentive reader may have noticed that technically, there is one input pair where the above algorithm returns the wrong result: When the two different encodings of 0, namely $10 \dots 00$ and $00 \dots 00$, are input. If we are computing $a < b$, the result will be wrong when $a = 10 \dots 00$ and $b = 00 \dots 00$ because a is treated as a negative number and b as a positive, returning 1, when in reality the correct output is 0. We can fix this by computing

$$fix = a_n \cdot (a_{n-1} + 1) \cdot \dots \cdot (a_1 + 1) \cdot (a_0) \cdot (b_n + 1) \cdot (b_{n-1} + 1) \cdot \dots \cdot (b_1 + 1) \cdot (b_0 + 1)$$

and adding it to res. The value of fix will be 1 exactly when $a = 10\ldots00$ and $b = 00\ldots00$ and 0 otherwise, so this corrects the wrong case. The additional effort for two n-bit numbers would be $2n-1$ additions, $2n-1$ multiplications, and a depth of $\lceil \log_2(2n) \rceil = \lceil \log_2(n)+1 \rceil$. However, in our computations, we mostly work with rational numbers (see Chapter 4) – depending on the parameters, it is highly unlikely that this will happen. For this reason, we stick with the above almost-correct algorithm, which has a much lower cost.
Switching from $a < b$ to $a \leq b$ can again be achieved by simply changing the subroutine `NatComp` appropriately as described in Section 3.1.2, at exactly the same cost. The original code is now correct also when $a = 10\ldots00$ and $b = 00\ldots00$, but incorrect in the other case, $a = 00\ldots00$ and $b = 10\ldots00$. We would thus change the fixing value to

$$fix = (a_n + 1) \cdot (a_{n-1} + 1) \cdot \cdots \cdot (a_1 + 1) \cdot (a_0) \cdot b_n \cdot (b_{n-1} + 1) \cdot \cdots \cdot (b_1 + 1) \cdot (b_0 + 1)$$

to correct this case if it is important.

3.3.4.1 *Effort*

To compare two n-bit numbers in Sign-Magnitude encoding, we have an effort of:

Line	**Field Additions**	**Field Multiplications**	**Depth**
3	$4n-4$	$n-1$	$n-1$
4	3	1	$+1$
Total	$4n-1$	n	n

The total cost with the fixing value would be $6n-2$ additions, $3n-1$ multiplications, and a depth of $\lceil \log_2(n)+1 \rceil$.

3.4 EFFORT COMPARISON

Since we have derived the efforts for different operations in Two's Complement (TC) and Sign-Magnitude (SM) encoding, we will compare them side by side through the formulas, graphically, and via some example values. For the runtimes, the implementation specifications can be found in Section 5.1.2.

3.4.1 Addition

We first recall the formulas for the effort of adding two n-bit numbers in the respective encodings:

	Field Additions	**Field Multiplications**	**Depth**
TC	$5n-2$	n	n
SM	$35n-60$	$8n-9$	$n+1$

We now graph this for different values of n in Figure 3.1.

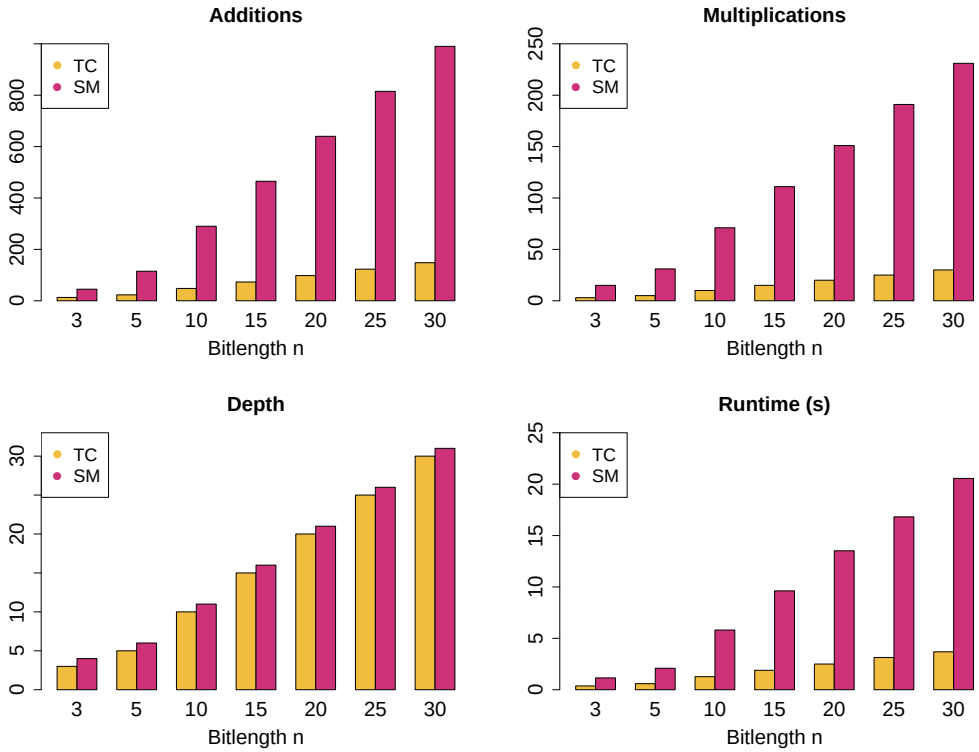

Figure 3.1: Number of bit additions, bit multiplications and multiplicative depth for adding two encrypted numbers of length n.

The concrete values can also been seen in Table 3.1.

Bitlength n	**TC Adds**	**SM Adds**	**TC Mults**	**SM Mults**	**TC Depth**	**SM Depth**	**TC Time (s)**	**SM Time (s)**
3	13	45	3	15	3	4	0.37	1.15
5	23	115	5	31	5	6	0.59	2.09
10	48	290	10	71	10	11	1.27	5.81
15	73	465	15	111	15	16	1.89	9.62
20	98	640	20	151	20	21	2.50	13.52
25	123	815	25	191	25	26	3.14	16.82
30	148	990	30	231	30	31	3.69	20.56

Table 3.1: Number of bit additions, bit multiplications and multiplicative depth for adding two encrypted numbers of length n.

We can see that Two's Complement is much more efficient than Sign-Magnitude in all aspects, which was to be expected from the formula.

3.4.2 Multiplication

We now compare the performance for multiplication in both encodings. We again first recall the formulas for the effort:

	Field Additions	**Field Multiplications**	**Depth**
TC	$\frac{5(m^2+n^2)-19(m+n)}{2} + 5mn + 10$	$m \cdot n + \frac{(n+m-2)\cdot(n+m-1)}{2}$	$m+n-1$
SM	$1 + \sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \lceil \frac{m-1}{2^i} \rceil \cdot \left(5 \cdot (n + 2^{i-1} + i) - 19\right)$	$\sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \lceil \frac{m-1}{2^i} \rceil \cdot \left(n + 2^{i-1} + i - 3\right)$	$m+n-3$

We again give some concrete values, choosing $m = n$ for comparability. The results can be seen graphically in Figure 3.2 and explicitly in Table 3.2.

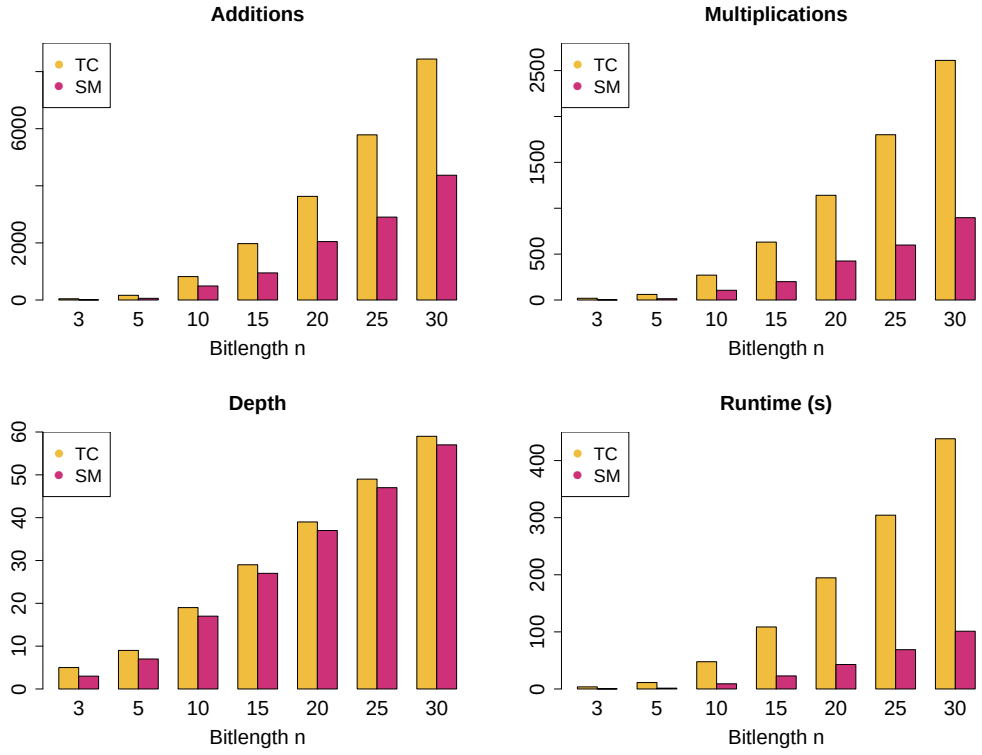

Figure 3.2: Number of bit additions, bit multiplications and multiplicative depth for multiplying two encrypted numbers of length n.

Bitlength n	TC Adds	SM Adds	TC Mults	SM Mults	TC Depth	SM Depth	TC Time (s)	SM Time (s)
3	43	7	19	2	5	3	3.68	0.21
5	165	59	61	14	9	7	11.22	1.44
10	820	491	271	106	19	17	47.72	9.03
15	1975	949	631	200	29	27	108.60	22.85
20	3630	2046	1141	425	39	37	194.64	42.85
25	5785	2904	1801	599	49	47	304.35	68.81
30	8440	4370	2611	897	59	57	438.18	101.14

Table 3.2: Number of bit additions, bit multiplications and multiplicative depth for multiplying two encrypted numbers of length n.

We can see that for multiplication, it is the other way around than for addition: Now, Sign-Magnitude encoding significantly outperforms Two's Complement encoding.

3.4.3 Negation

First, recall the formulas for the effort from Sections 3.2.3 and 3.3.3:

	Field Additions	Field Multiplications	Depth
TC	$6n - 7$	$n - 1$	$n - 1$
SM	1	0	0

We can already see that Sign-Magnitude is much better here, but we still present the results in Figure 3.3 and in Table 3.3.

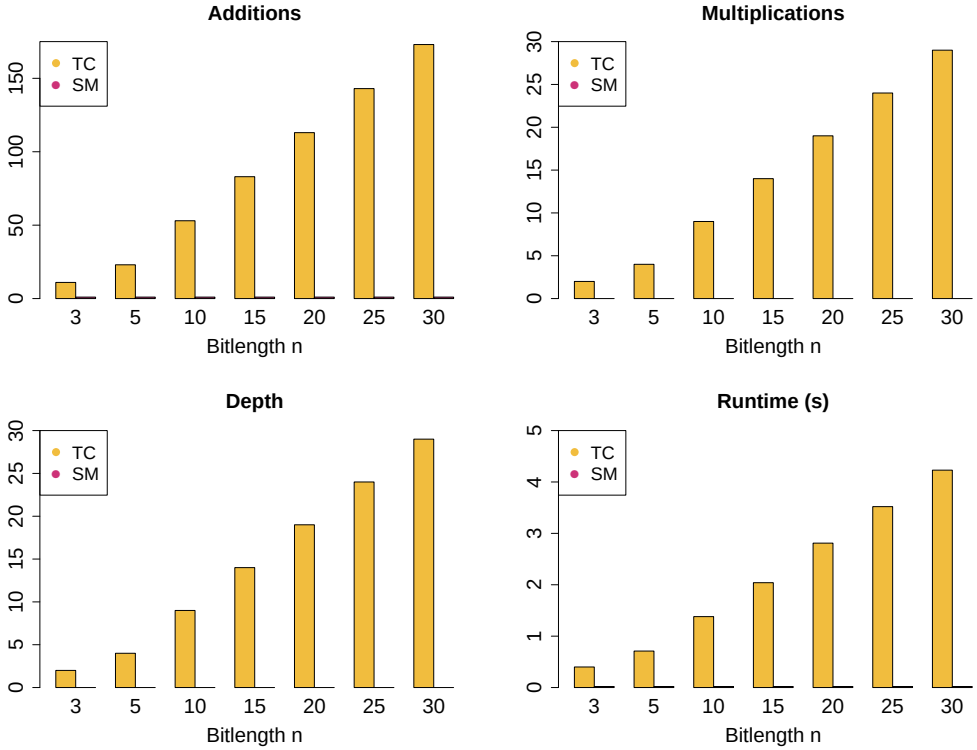

Figure 3.3: Number of bit additions, bit multiplications and multiplicative depth for negating an encrypted number of length n.

Bitlength n	**TC Adds**	**SM Adds**	**TC Mults**	**SM Mults**	**TC Depth**	**SM Depth**	**TC Time (s)**	**SM Time (s)**
3	11	1	2	0	2	0	0.40	0.02
5	23	1	4	0	4	0	0.71	0.02
10	53	1	9	0	9	0	1.38	0.02
15	83	1	14	0	14	0	2.04	0.02
20	113	1	19	0	19	0	2.81	0.02
25	143	1	24	0	24	0	3.52	0.02
30	173	1	29	0	29	0	4.23	0.02

Table 3.3: Number of bit additions, bit multiplications and multiplicative depth for negating a number of length n.

3.4.4 Comparison

Lastly, we also look at the costs of comparing two n-bit numbers in each encoding. Recall that for Sign-Magnitude, we use the cheaper comparison where the input can sometimes be wrong if the two different encodings of 0 are compared. The efforts for comparing two n-bit numbers from Sections 3.2.4 and 3.3.4 are:

	Field Additions	**Field Multiplications**	**Depth**
TC	$4n + 4$	$n + 2$	$n + 1$
SM	$4n - 1$	n	n

We present some example values in Figure 3.4 and in Table 3.4.

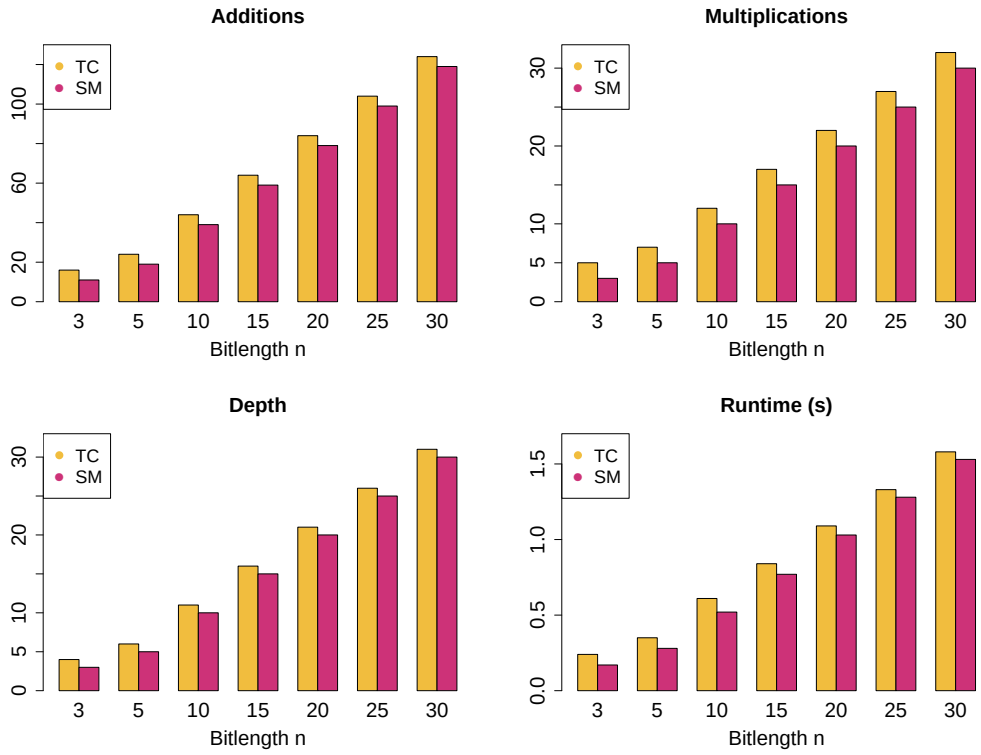

Figure 3.4: Number of bit additions, bit multiplications and multiplicative depth to compare two encrypted numbers of length n.

Bitlength n	**TC Adds**	**SM Adds**	**TC Mults**	**SM Mults**	**TC Depth**	**SM Depth**	**TC Time (s)**	**SM Time (s)**
3	16	11	5	3	4	3	0.24	0.17
5	24	19	7	5	6	5	0.35	0.28
10	44	39	12	10	11	10	0.61	0.52
15	64	59	17	15	16	15	0.84	0.77
20	84	79	22	20	21	20	1.09	1.03
25	104	99	27	25	26	25	1.33	1.28
30	124	119	32	30	31	30	1.58	1.53

Table 3.4: Number of bit additions, bit multiplications and multiplicative depth to compare two encrypted numbers of length n.

We see that to compare two numbers, Sign-Magnitude encoding performs slightly better than Two's Complement encoding – however, this difference is almost negligible, as the two only differ by very small additive factors. If we were to use the fixed circuit for Sign-Magnitude, we would find that Two's Complement performs slightly better.

3.4.5 Conclusion

In conclusion, we have seen that for comparison, there is very little difference between Two's Complement encoding and Sign-Magnitude encoding. Two's Complement costs a lot less than Sign-Magnitude if the task is to add two numbers, but Sign-Magnitude is the better choice for multiplying two numbers or negating a number. This motivates the next section, where we will combine both encodings to utilize this performance difference.

3.5 HYBRID ENCODING

Since we have seen that Two's Complement encoding performs much better for addition, but Sign-Magnitude is much more efficient for multiplication, we now propose a mix of the two, called Hybrid Encoding. The idea is to use Two's Complement encoding, but when we want to multiply, we switch to Sign-Magnitude encoding, perform the multiplication there, and then switch back to Two's Complement for subsequent operations. This switching of course induces some costs of its own – however, we will see that for real-world applications, it performs better than each encoding on its own. However the overhead is big enough that we would not switch for comparison or negation, especially since the latter is part of the switching procedure, which we will present in **Section 3.5.1**. We examine the performance for individual operations in **Section 3.5.2**, and the performance in real-world problems from the area of Machine Learning will be seen in Chapter 5.

3.5.1 Switching

We now explain how to get from one encoding to the other. First, note that for positive numbers (i.e., $a_n = 0$), the two encodings are actually the same: We have

$$a = a_n \cdot (-2^n) + \sum_{i=0}^{n-1} a_i \cdot 2^i = 0 \cdot (-2^n) + \sum_{i=0}^{n-1} a_i \cdot 2^i = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

in Two's Complement encoding, and

$$a = (-1)^{a_n} \cdot \sum_{i=0}^{n-1} a_i \cdot 2^i = (-1)^0 \cdot \sum_{i=0}^{n-1} a_i \cdot 2^i = 1 \cdot \sum_{i=0}^{n-1} a_i \cdot 2^i = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

in Sign-Magnitude encoding.
Thus, if the number is positive, we do not have to do anything. If the number is negative, we can negate it in the current encoding ($a \mapsto -a$), so that it is now a positive number and thus the same in both encodings. Negating it again in the new encoding maps the input back to its original value, but in the other encoding. We call the negation operations `TCNeg` (see Section 3.2.3) and `SMNeg` (see Section 3.3.3), respectively. Then, to convert a negative number a from Two's Complement to Sign-Magnitude, we would compute

$$a_{TC} \overset{\texttt{TCNeg}}{\mapsto} -a_{TC} = -a_{SM} \overset{\texttt{SMNeg}}{\mapsto} a_{SM}.$$

Of course, because the number is encrypted, we do not know whether the number is positive or not – thus, we compute both cases and combine them by multiplexing on the conditional a_n, the sign bit. The exact workings can be found in Algorithm 6 for switching from Two's Complement to Sign-Magnitude, and in Algorithm 7 for switching from Sign-Magnitude to Two's Complement.

Algorithm 6: `Switching Two's Complement to Sign-Magnitude`

Input: Signed number $a = a_n \dots a_1 a_0$ in Two's Complement encoding

`// Get the negative of` a `in Two's Complement encoding.`

1 $\tilde{a} = \texttt{TCNeg}(a)$

`// Get the negative of` $\tilde{a}$ `in Sign-Magnitude encoding.`

2 $\bar{a} = \texttt{SMNeg}(\tilde{a})$

`// Assign` $\bar{a}$ `to the result if` a `is negative (`$a_n = 1$`), and assign` a `otherwise.`

3 $res = \texttt{MUX}(a_n, \bar{a}, a)$

Output: res

We can see that the costs we get from switching are that of one Two's Complement negation, one Sign-Magnitude negation, and one `MUX` on n-bit numbers. We present the cost for Algorithm 6, but it is obviously exactly the same for Algorithm 7:

Line	Field Additions	Field Multiplications	Depth
1	$6n - 7$	$n - 1$	$n - 1$
2	1	0	0
3	$2n$	n	$+1$
Total	$8n - 6$	$2n - 1$	n

Algorithm 7: `Switching Sign-Magnitude to Two's Complement`

Input: Signed number $a = a_n \dots a_1 a_0$ in Sign-Magnitude encoding
`// Get the negative of` a `in Sign-Magnitude encoding.`
1 $\tilde{a} = \texttt{SMNeg}(a)$
`// Get the negative of` $\tilde{a}$ `in Two's Complement encoding.`
2 $\bar{a} = \texttt{TCNeg}(\tilde{a})$
`// Assign` $\bar{a}$ `to the result if` a `is negative (`$a_n = 1$`), and assign` a `otherwise.`
3 $res = \texttt{MUX}(a_n, \bar{a}, a)$
Output: res

We can already asymptotically see that this will likely be more efficient than Sign-Magnitude multiplication for large enough bitlengths, as the switching adds a cost linear in the input length, and the two multiplication algorithms have costs quadratic in the input lengths and differ greatly. We will see some concrete numbers in the next section.

3.5.2 Performance

We now examine the effects of replacing the Two's Complement multiplication with our new procedure. Denoting the Sign-Magnitude multiplication of two numbers `SMMult` and the switching procedures with `SwitchTCSM` (Two's Complement to Sign-Magnitude, Algorithm 6) and `SwitchSMTC` (Sign-Magnitude to Two's Complement, Algorithm 7), the workings can be seen in Algorithm 8.

Algorithm 8: `Hybrid Encoding Multiplication`

Input: Signed number $a = a_n \dots a_1 a_0$ in Two's Complement encoding
Input: Signed number $b = b_m \dots b_1 b_0$ in Two's Complement encoding
`// Switch` a `and` b `to Sign-Magnitude encoding.`
1 $\tilde{a} = \texttt{SwitchTCSM}(a)$
2 $\tilde{b} = \texttt{SwitchTCSM}(b)$
`// Multiply the values.`
3 $temp = \texttt{SMMult}(\tilde{a}, \tilde{b})$
`// Switch` $temp$ `back to Two's Complement encoding.`
4 $res = \texttt{SwitchSMTC}(temp)$
Output: res

The effort we get from this (where *Switching Overhead* means everything except the actual multiplication costs, i.e., the costs we have on top of the Sign-Magnitude multiplication due to the switching procedures) is:

Line	**Field Additions**	**Field Multiplications**	**Depth**
1	$8n-6$	$2n-1$	n
2	$8m-6$	$2m-1$	m
3	$1+\sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \left(\lceil \frac{m-1}{2^i} \rceil \cdot \left(5 \cdot (n+2^{i-1}+i)-19\right) \right)$	$\sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \left(\lceil \frac{m-1}{2^i} \rceil \cdot \left(n+2^{i-1}+i-3\right) \right)$	$m+n-3$
4	$8(m+n)-6$	$2(m+n)-1$	$m+n$
Switching Overhead	$16(m+n)-18$	$4(m+n)-3$	$m+n$
Total	$\sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \left(\lceil \frac{m-1}{2^i} \rceil \cdot \left(5 \cdot (n+2^{i-1}+i)-19\right) \right) +16(m+n)-17$	$\sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \left(\lceil \frac{m-1}{2^i} \rceil \cdot \left(n+2^{i-1}+i-3\right) \right) +4(m+n)-3$	$m+n$

Note that for depth, we again have the effect that the depth does not increase by performing these sequential operations. As before, this is because the input bits in the addition subroutines have the same depth as the carry for that position (as opposed to depth 0 when adding two fresh ciphertexts) and are merely added to the carry, so the depth of each digit does not change compared to when the inputs are two fresh ciphertexts where each bit has depth 0.

We again present some example values for the multiplication in Figure 3.5 and Table 3.5. Note that addition is just Two's Complement addition, so for this we refer the reader back to Section 3.4.1.

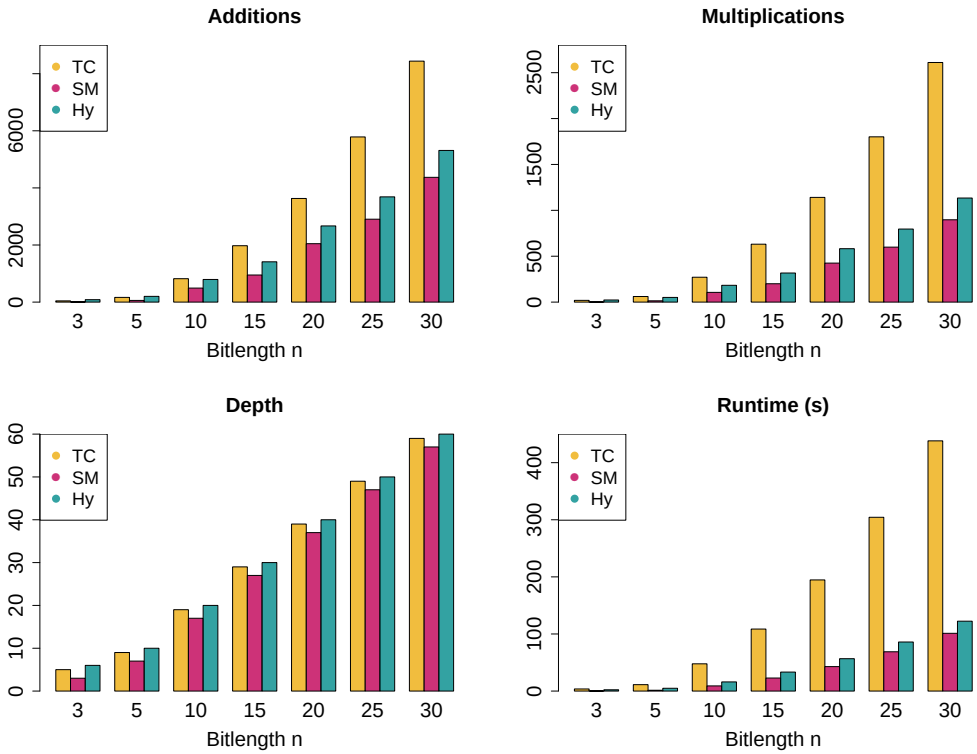

Figure 3.5: Number of bit additions, bit multiplications, multiplicative depth and runtime to multiply two encrypted numbers of length n.

Bitlength n	**TC Adds**	**SM Adds**	**Hybrid Adds**	**TC Mults**	**SM Mults**	**Hybrid Mults**
3	43	7	85	19	2	23
5	165	59	201	61	14	51
10	820	491	793	271	106	183
15	1975	949	1411	631	200	317
20	3630	2046	2668	1141	425	582
25	5785	2904	3686	1801	599	796
30	8440	4370	5312	2611	897	1134

Bitlength n	**TC Depth**	**SM Depth**	**Hybrid Depth**	**TC Time(s)**	**SM Time(s)**	**Hybrid Time(s)**
3	5	3	6	3.68	0.21	2.26
5	9	7	10	11.22	1.44	4.89
10	19	17	20	47.72	9.03	16.03
15	29	27	30	108.60	22.85	33.25
20	39	37	40	194.64	42.85	56.70
25	49	47	50	304.35	68.81	85.92
30	59	57	60	438.18	101.14	122.38

Table 3.5: Number of bit additions, bit multiplications, multiplicative depth and runtime to multiply two encrypted numbers of length n.

We see that even for relatively small bitlengths, the new multiplication algorithm significantly outperforms Two's Complement multiplication both in theory (as illustrated by the theoretical values for the number of additions and multiplications in Figure 3.5 and Table 3.5) and in terms of actual runtimes, at the cost of only increasing overall depth by 1. Recall that the multiplication algorithm is in essence that of Sign-Magnitude multiplication, but with the additional cost of switching. Thus, it is obvious that Sign-Magnitude performs better for multiplication alone – however, Hybrid Encoding lets us use the more efficient Two's Complement addition, so we can have the best of both worlds in computations that involve both additions and multiplications.

3.6 CONCLUSION

In conclusion, we have examined the costs of the two most popular binary encodings that allow for negative integers, and have shown that one (Two's Complement) is better for addition, and the other (Sign-Magnitude) is better for multiplication. To utilize the respective benefits, we came up with a new encoding, which essentially replaces the costly Two's Complement multiplication algorithm by switching to Sign-Magnitude for multiplication, and then switching back to Two's Complement for addition. We have seen that this new encoding, called Hybrid Encoding, vastly outperforms the Two's Complement multiplication, and in Chapter 5 we will further see the impact when using more complicated real-world functions from the realm of Machine Learning.

Chapter 4

RATIONAL NUMBERS

In this chapter, we show how to incorporate rational numbers into FHE computations. To this end, we first present an encoding in **Section 4.1** that in theory even allows the division of two encrypted numbers, which is not usually possible. We do this by encoding rational numbers as fractions with separately encrypted numerator and denominator. We also discuss the downsides of this encoding, and instead opt to use scaling to map rationals to integers. In this context, we also show how to keep the precision constant in **Section 4.2**. We then present some other improvements in **Section 4.3**, like accelerated comparison variants and a method for the management of the bitlength under certain assumptions. This chapter is largely taken from [JA16] and [JA18].

4.1 FRACTIONAL ENCODING

We present a new encoding for rational numbers, called *Fractional Encoding*, which builds on the encodings for integers presented so far but allows the division of two encrypted ciphertexts, which is not normally possible in Fully Homomorphic Encryption. The idea is to express the number we wish to encode as a fraction and encode the numerator and denominator separately. There are many ways to do this, but it is important that the underlying numbers not be too small, as will be explained in Section 4.1.3. Thus, if we were to encode a number like $1/4$, one should instead use numerator and denominator like $10000/40000$, depending on the parameters. The generic method we use to encode a number a is to choose the denominator a_{den} randomly in a certain range (like $a_{den} \in [2^k, 2^{k+1})$ for some k) and compute the numerator a_{num} as $a_{num} = \lfloor a \cdot a_{den} \rceil$. We then encode and encrypt both separately, so we have $a = (a_{num}, a_{den})$ with numerator and denominator encoded bitwise as in the previous chapters.

Example 4.1: *Suppose we want to encode the number* $a = 1.542$*. Then we could pick* $k = 15$*, i.e., we choose the denominator randomly in the range* $[32768, 65536)$*. Say we choose the denominator as* $a_{den} = 55555$*, then we compute the numerator as* $a_{num} = a \cdot a_{den} = 85665.81$*, which we round to* 85666*. Thus, the number* a *is encoded as*

$$a = (85666, 55555) = (0101001110101000 10, 01101100100000011).$$

4.1.1 Operations

If we want to perform computations (including division) on values encoded in this way, we can express the operations using the subroutines from the binary encoding for integers through the regular computation rules for fractions. We present these operations in the following, where `Add`, `Mult` and `Comp` denote the regular routines from the Hybrid Encoding of Section 3.5, and `FracAdd`, `FracMult`, `FracDiv` and `FracComp` denote the respective operations in the Fractional Encoding.

- $a + b$: Recall that

$$\frac{a_{num}}{a_{den}} + \frac{b_{num}}{b_{den}} = \frac{a_{num} \cdot b_{den} + b_{num} \cdot a_{den}}{a_{den} \cdot b_{den}}.$$

  Thus, we get

$$\begin{aligned}&\texttt{FracAdd}((a_{num}, a_{den}), (b_{num}, b_{den}))\\ &= \big(\texttt{Add}(\texttt{Mult}(a_{num}, b_{den}), \texttt{Mult}(a_{den}, b_{num})), \texttt{Mult}(a_{den}, b_{den})\big).\end{aligned}$$

- $a \cdot b$: We have

$$\frac{a_{num}}{a_{den}} \cdot \frac{b_{num}}{b_{den}} = \frac{a_{num} \cdot b_{num}}{a_{den} \cdot b_{den}},$$

  so

$$\texttt{FracMult}((a_{num}, a_{den}), (b_{num}, b_{den})) = \big(\texttt{Mult}(a_{num}, b_{num}), \texttt{Mult}(a_{den}, b_{den})\big).$$

- a/b : It holds that

$$\frac{a_{num}}{a_{den}} \Big/ \frac{b_{num}}{b_{den}} = \frac{a_{num} \cdot b_{den}}{a_{den} \cdot b_{num}},$$

  so

$$\texttt{FracDiv}((a_{num}, a_{den}), (b_{num}, b_{den})) = \big(\texttt{Mult}(a_{num}, b_{den}), \texttt{Mult}(a_{den}, b_{num})\big).$$

- $a \leq b$: This is slightly more involved. To see this, consider the following: We basically want to compare $\frac{a_{num}}{a_{den}}$ and $\frac{b_{num}}{b_{den}}$, so instead of asking whether

$$\frac{a_{num}}{a_{den}} \leq \frac{b_{num}}{b_{den}},$$

  we multiply with the denominators and ask whether

$$a_{num} \cdot b_{den} \leq b_{num} \cdot a_{den}.$$

  However, multiplying an inequality with a negative value changes the direction of the inequality. Thus, if the sign of exactly one of the denominators is negative, this changes the direction of the inequality operator, so that we would need to compute

$$b_{num} \cdot a_{den} \leq a_{num} \cdot b_{den}$$

  instead. We thus assign the values conditionally through a multiplexer gate before comparing them: If the `XOR` of the sign bits is 0 (i.e., the signs are equal), we compare

$$(e := a_{num} \cdot b_{den}) \leq (d := b_{num} \cdot a_{den}),$$

and if it is 1, i.e., exactly one of the denominators is negative, we compare

$$(e := b_{num} \cdot a_{den}) \leq (d := a_{num} \cdot b_{den}).$$

Formally, recall that the MSB determines the sign of the number (1 if it is negative and 0 otherwise). Let

$$c := \text{Sign}(a_{den}) \oplus \text{Sign}(b_{den}) = \begin{cases} 0 & \text{if the signs are equal} \\ 1 & \text{if the signs are unequal.} \end{cases}$$

Then we set

$$d := \texttt{MUX}(c, \texttt{Mult}(a_{num}, b_{den}), \texttt{Mult}(a_{den}, b_{num}))$$

and

$$e := \texttt{MUX}(c, \texttt{Mult}(a_{den}, b_{num}), \texttt{Mult}(a_{num}, b_{den}))$$

and output the result as $\texttt{Comp}(e, d)$.

Thus, we can write:

$$\begin{aligned} &\texttt{FracComp}((a_{num}, a_{den}), (b_{num}, b_{den})) \\ =&\texttt{Comp}(e, d) \\ =&\texttt{Comp}\Big(\texttt{MUX}(c, \texttt{Mult}(a_{den}, b_{num}), \texttt{Mult}(a_{num}, b_{den})), \\ &\qquad \texttt{MUX}(c, \texttt{Mult}(a_{num}, b_{den}), \texttt{Mult}(a_{den}, b_{num}))\Big). \end{aligned}$$

Note that if we accept that the result is wrong in the case $a = b$ in our comparison, we could save some effort by not multiplexing but instead computing

$$\texttt{FracComp}((a_{num}, a_{den}), (b_{num}, b_{den})) = \texttt{Comp}(\texttt{Mult}(a_{num}, b_{den}), \texttt{Mult}(a_{den}, b_{num})) \oplus c.$$

This flips the result of the original comparison $a_{num} \cdot b_{den} \leq b_{num} \cdot a_{den}$ if the sign bits are unequal – in other words, it changes $a_{num} \cdot b_{den} \leq b_{num} \cdot a_{den}$ to $a_{num} \cdot b_{den} > b_{num} \cdot a_{den}$ rather than $a_{num} \cdot b_{den} \geq b_{num} \cdot a_{den}$ as would be correct. However, in our computations in Section 5.4, we will use the correct way described above.

4.1.2 Controlling the Bitlength

Notice that every single one of the operations above requires a multiplication of some sort, which means that the bitlengths of the numerators and denominators doubles with each operation because there is no cancellation when the data is encrypted. However, we can solve this problem using the following Lemma:

Lemma 9. *Given a number a and its binary representation $a_{n-1} \dots a_1 a_0$, deleting the last k bits corresponds to dividing by a by 2^k and truncating the result.*

Proof: Let $a = \sum_{i=0}^{n-1} a_i 2^i$ be encoded through its bits $a_{n-1} \dots a_1 a_0$. Then removing the last k bits leaves us with $a^T := a_{n-1} \dots a_{k+1} a_k$. This evaluates to

$$\sum_{i=k}^{n-1} a_i 2^{i-k} = \frac{2^k}{2^k} \sum_{i=k}^{n-1} a_i 2^{i-k} = \frac{1}{2^k} \sum_{i=k}^{n-1} a_i 2^i = \frac{1}{2^k}(a - \sum_{i=0}^{k-1} a_i 2^i) = \frac{a}{2^k} - \sum_{i=0}^{k-1} a_i \frac{2^i}{2^k}.$$

Now note that the sum being subtracted in the rightmost part is exactly the terms with a negative power of 2 – in other words, the coeffients that come behind the binary point. Thus the end result is the integer that we get by disregarding anything past the binary point, which is exactly what truncation does.

□

Thus, deleting the last k least significant bits for both numerator and denominator yields roughly the same result as before, but with lower bitlengths.

Example 4.2: *Suppose that we have encoded our integers with* 15 *bits, and after multiplication we thus have* 30 *bits in numerator and denominator, e.g.* $651049779/1053588274 \approx 0.617936$. *Then dividing both numerator and denominator by* 2^{15} *and truncating yields* $19868/32152$, *which evaluates to* $0.617939 \approx 0.617936$. *The accuracy can be set through the original encoding bitlength (*15 *here).* ———

4.1.3 Performance

While we will evaluate the performance of this encoding on a real-world example in Section 5.4, we already discuss some inherent problems that we see with this encoding at this point.

The first issue with this encoding is the runtime. Even though the library we use, [LIBf], is currently the most efficient FHE library with which many computational tasks approach practically feasible runtimes, the fact that this encoding requires several multiplications on binary numbers for each elementary operation slows it down considerably and makes computation times prohibitively large.

The second issue encountered in Fractional Encoding is the procedure to shorten the bitlengths that was described in Section 4.1.2. While it works reasonably well for short computations, we found it nearly impossible to set the number of bits to delete such that a more involved algorithm ran correctly. The reason is simple: If not enough bits are cut off, the bitlength grows, propagating with each operation and resulting in an overflow when the number becomes too large for the allocated bitlength. If too many bits are cut off, one loses too much accuracy or may even end with a 0 in the denominator. Both these cases result in completely arbitrary and unusable results. The reason why it is so hard to set the shortening parameter properly is that generally, numerator and denominator will not require the same number of bits. Concretely, this will only be the case when the value of the number being in encoded is in the interval $(1/2, 2)$, and even so, this interval is not closed under addition or multiplication, so this problem can arise even if plaintexts are scaled into this range.

Example 4.3: *As an extreme example, suppose we encoded our plaintexts as 10-bit numbers and want to compute* `FracMult`$((45, 555), (91, 600))$. *Then we get a result of* $(4095, 333000) \approx 0.0123$, *which we can now shorten. Suppose we shorten by* 10, *i.e., the original precision: Then we get* $(3, 325) \approx 0.009$, *which does not seem too bad. However, note that the numerator is extremely small now – imagine dividing this number by itself (in lieu of dividing by another number with similar structure), which should result in* 1. *Instead, we get* $(975, 975)$, *which after shortening becomes* $(0, 0)$, *essentially ruining any further computation in which it is involved because the result will always be* $(0, 0)$.
If, on the other hand, we chose to only shorten by 8 *bits instead of* 10, *the result of our multiplication would be* $(15, 1300) \approx 0.0115$. *However, the denominator now requires* 11 *bits to encode. Squaring this number (in lieu of multiplying with a number of similar structure) results in* $(225, 1690000)$, *which after shortening by* 8 *bits becomes* $(0, 6601)$. *We can argue that the* 0 *nominator is close enough to the actual result, but the denominator now requires* 13 *bits. This will continue until the bitlength exceeds the allocated amount and an overflow occurs, so unless more space is allocated with each operation (which leaks the number of operations performed and may be undesirable, not to mention extremely inefficient), the computation will fail.* ———

The problem is that because the data is encrypted, we cannot see the actual size of the underlying data, so the shortening parameter cannot be set dynamically – in fact, if it were possible to set it dynamically, this would imply that the FHE scheme is insecure. Also, even setting the parameter roughly requires extensive knowledge about the encrypted data, which the data owner may not want to share with the computing party.
In conclusion, this encoding is theoretically possible, but we would not recommend it for practical use due to its inefficiency and hardness of setting the shortening parameter (or even higher inefficiency if little to no shortening is done). We still compare its performance to other solutions when evaluating the K-Means-Algorithm in Section 5.4 to see the real-world performance of this encoding, but the downfalls of this encoding are what led us to modify the K-Means-Algorithm in Section 5.4.4 to avoid division and be able to use the scaling variant in the first place. We note, however, that for very flat computations (in the sense that there are not many successive operations performed), this encoding that allows division may still be of interest.

4.2 SCALING

To avoid the use of this fractional encoding, we now instead show how to map rational numbers to integers by scaling. In previous work (e.g. [AEH15]), rational numbers have often been approximated by multiplying with a power of some number, called the *scaling factor* (e.g. 10 or 2), and rounding to obtain an integer. However, note that generally when multiplying two rational numbers with k bits of precision, we obtain a number with $2k$ bits of precision (whereas addition does not change the precision). If we were working on unencrypted numbers, we might just round to obtain k bits of precision again, or we could truncate (truncation after k bits yields the same accuracy as rounding to $k - 1$ bits). However, things are more difficult when operating on encrypted data, as rounding is generally not possible here and thus these extra bits of precision accumulate. To see this, suppose we would like to work with a precision of k bits, and the scaling factor w. This means that we multiply a rational number with w^k and round (or truncate)

to the nearest integer, which is then encoded and encrypted as in the previous sections. Dividing the decrypted decoded number by w^k again yields the rounded rational. However, the problem of doubling precision with multiplication is prevalent here. Consider what would happen if we were to multiply two such numbers: Suppose we have two rational numbers a and b that we would like to encode as integers a' and b' with k digits of precision, so we have $a' = \lceil a \cdot w^k \rfloor$ and $b' = \lceil b \cdot w^k \rfloor$. Multiplying a' and b', we get $c'' = a' \cdot b' \approx a \cdot w^k \cdot b \cdot w^k = (a \cdot b) \cdot w^{2k}$. Thus, having reversed the encoding, the obtained value c'' must be divided by w^{2k}. This is a problem because we cannot remove the extra bits by dividing by w^k, so the party performing the algorithm must now divulge what power of w to divide the obtained result by after decryption. This leaks information about the multiplicative depth of the function used and thus constitutes a privacy breach for the computing party. Additionally, there is also the problem during computation that the sizes of the encoded numbers will increase substantially.

Example 4.4: *Let $a = 1.342$ and $b = 4.11$. We pick a precision of $k = 2$ and a scaling factor of $w = 10$, so we get the integers*

$$a' = \lfloor 1.342 \cdot 10^2 \rceil = 134 \quad \textit{and} \quad b' = \lfloor 4.11 \cdot 10^2 \rceil = 411.$$

Multiplying these two numbers, we get $134 \cdot 411 = 55074$. To get the right result, we now have to divide by $(10^2)^2 = 10000$, yielding $5.5074 \approx 1.342 \cdot 4.1 = a \cdot b$. Note that the result has 4 digits of precision even though we encoded the inputs with only 2 digits of precision, and thus requires a larger bitlength. ———

Recall from Lemma 9 in Section 4.1.2 that deleting the last k bits corresponds to dividing by 2^k and truncating the result. To utilize this fact, we propose the following approach: Instead of scaling by a power of an arbitrary number w, we set $w = 2$ and truncate to obtain an integer. As in the previous chapters, we encode this integer in binary fashion, so that we can encrypt each bit separately. This eliminates the above problem: Multiplying two numbers a' and b' with k bits of precision still yields $c'' = (a \cdot b) \cdot 2^{2k}$, but since we are encoding bit by bit, we can manipulate the individual bits and thus simply delete the last k (encrypted) bits of the product, which corresponds to dividing by 2^k and truncating as per Lemma 9. This way, the party performing the computations can bring the product c'' back down to the required precision after every step by discarding the last k bits and thus obtaining $c' = a \cdot b \cdot 2^k$. This solves the above problem because the party which holds the data must now always divide the decoded result by 2^k no matter what operations were applied. This has the benefit of not only hiding the data from the computing party, but also hiding the function from the party with the data while simultaneously avoiding superfluous bitlength. The exact workings can be seen in Algorithm 9.

Algorithm 9: `Multiplication using scaling`

`Encoding():`

Input: A rational number a
Input: The scaling factor k
`// Scale and truncate.`
1 $a' := \lfloor 2^k \cdot a \rfloor$
`// Encode with Two's Complement or Sign-Magnitude or Hybrid Encoding.`
2 $\tilde{a} := \texttt{Encode}(a')$
Output: $\tilde{a}$

`Multiplication():`

Input: A number $\tilde{a}$ as encoded above
Input: A number $\tilde{b}$ as encoded above
Input: The scaling factor k (must be the same for $\tilde{a}$ and $\tilde{b}$)
`// Multiply using bitwise algorithm of choice`
3 $c = c_n \dots c_1 c_0 := \texttt{Mult}(\tilde{a}, \tilde{b})$
`// Delete last` k `bits`
4 $\tilde{c} := c_n \dots c_{k+1} c_k = d_{n'} \dots d_1 d_0$
Output: $\tilde{c}$

`Decoding():`

Input: A number $d_{n'} \dots d_1 d_0$ as encoded above
Input: The scaling factor k
`// Obtain integer`
5 $d' = \sum_{i=0}^{n'} d_i 2^i$
`// Divide by scaling factor`
6 $d := d'/2^k$
Output: d

To make the algorithm clearer, consider the following example:

Example 4.5: *We return to the above example with* $a = 1.342$ *and* $b = 4.11$*. We pick a precision of* $k = 7$*, so we get the integers*

$$a' = \lfloor 1.342 \cdot 2^7 \rceil = 172 \quad \textit{and} \quad b' = \lfloor 4.11 \cdot 2^7 \rceil = 526.$$

Multiplying these two numbers, we get $172 \cdot 526 = 90472$*. To get the right result, we would now have to divide by* $(2^7)^2 = 16384$*. However, we are operating bitwise, so we have the result (in encrypted form) as* $90472 = 10110000101101000$*.*
Deleting the last $k = 7$ *bits, we are left with* $c' = 1011000010$*, which encodes the number* 706*. If the user receives this encrypted result he can divide be the original scaling factor* 2^7 *after decrypting to obtain the final value* $706/(2^7) \approx 5.52$*. Thus, the user has obtained the correct result without needing any further information that depends on the applied function in order to decode correctly.* ———

Note also that the result c' in the above example now only has $k = 7$ bits of precision instead of $2 \cdot k = 14$ as it would traditionally. This is especially important if the value is an input to further computations, as we have seen in Chapter 3 that most elementary functions are linear or quadratic in the input length, so shorter lengths mean faster computation.

4.2.1 Relation to Floating Point Representation

As our scaling approach is somewhat similar to the floating point representation used to represent rational numbers in unencrypted computations, we briefly discuss differences and similarities.
In floating point representation, a number is represented by a significand s and and exponent e relative to some base b. The number is then computed as $s \cdot b^e$, which of course looks very similar to our scaling encoding. However, in floating point representation, the exponent is explicitly part of the representation and involved in the computations: To add two numbers in different representations, one scales the smaller number to have the larger exponent as well (e.g., in base 10, we would scale $13.21 = 1.321 \cdot 10^2$ to $13.21 = 0.001321 \cdot 10^5$ to get from exponent 2 to exponent 5) and then adds the two significands. To multiply two numbers, the significands are multiplied and the exponents are added.
Thus, for floating point computations, the exponents would have to be encrypted as well as the significands, and especially scaling to the same exponent for addition may prove complex to do homomorphically. If we do not encrypt the exponents, they may leak information about the operations that were performed. Either way, this approach also suffers from the issue of doubling precision with each multiplication, though we could likely manage this by an approach similar to our above one.
In essence, our representation is like the floating point representation, but with a fixed exponent that stays the same throughout all computations, instead introducing more variability into the significand. True floating point representation would be somewhat complex to perform on encrypted numbers, but a combination with our approach (i.e., leaving the exponents unencrypted, but reducing precision to hide what computation was performed) would likely be feasible, though a higher efficiency than our approach is improbable due to the more complex nature of the computations.

4.3 OTHER IMPROVEMENTS

We now showcase some other improvements that can reduce runtimes under certain assumptions. These methods are not per se restricted to rational numbers, but for maximum generality we include them in this chapter.

4.3.1 Easy Comparison

Recall that we have derived elaborate comparison functions in Sections 3.2.4 and 3.3.4. However, there are numerous use cases where one only has to compare a number to 0, like in the Perceptron in Section 5.3, and in this case there is a much easier way: Instead of computing a costly circuit for comparison, it suffices to take the most significant bit of the number, which will be 0 if the number is greater than zero and 1 if it is less. For Two's Complement, it will be 0 also when the number equals 0, but in Sign-Magnitude it can be either 0 or 1 when using this method, as we recall that there are two encodings of 0 here. Thus, if what we are comparing is exactly 0, the resulting bit is wrong for Two's Complement and can be either wrong or right for Sign-Magnitude. We observe, however, that when working with rationals of high enough precision, many functions are highly unlikely to be 0. For example, a weighted sum $w_1x_1 + \cdots + w_lx_l$ is unlikely to be exactly 0 if the weights $w_1, \ldots, w_l$ are random rational numbers. Thus, in this case there should be no change whether the condition for an operation is $w_1x_1 + \cdots + w_lx_l > 0$ or $w_1x_1 + \cdots + w_lx_l \geq 0$, and the easy comparison should return the correct result with overwhelming probability. Of course, if the coefficient or variable domain space is very small (e.g., integers in some range) in the more general case, a more involved formula should be used.

4.3.2 Approximate Comparison

Our second improvement is also concerned with the comparison function, but in the context of iterative algorithms that converge over time, like the K-Means-Algorithm that we will examine in Section 5.4. The idea is that in the beginning, there is a lot of change, and towards the end, the amount of change decreases as we near the final value. If comparisons are involved in this change, then in the beginning, it will likely (though this of course depends on the concrete application) not greatly influence the behaviour of the algorithm if the comparison is occasionally wrong when the difference between the inputs is very small, but as we converge towards the final value, these small differences may become more important.

Thus, we now present the following modification which trades in a bit of accuracy for slightly improved runtime: Since the `Compare` function is linear in the length of its inputs, speeding up this building block would make the entire computation more efficient. To do this, first recall that because we encode our numbers in a bitwise fashion after scaling them to integers, we have access to the individual bits and can, for example, delete the S least significant bits, which corresponds to dividing the number by 2^S and truncating. Let $\tilde{a}$ denote this truncated version of a number a, and $\tilde{b}$ that of a number b. Then $\texttt{Compare}(\tilde{a}, \tilde{b}) = \texttt{Compare}(a, b)$ if $|a - b| \geq 2^S$, and may or may not return the correct result if $|a - b| < 2^S$. However, correspondingly, if the result is wrong, the difference between the two inputs is no more than 2^S. The exact workings of this approximate

comparison, denoted `ApproxCompare`, can be seen in Algorithm 10.

Algorithm 10: `ApproxCompare`(a, b, S)

Input: The two arguments a, b, encoded bitwise
Input: The accuracy factor S
`// Corresponds to` $\tilde{a} = \lfloor a/2^S \rfloor$
1 Remove last S bits from a, denote $\tilde{a}$
`// Corresponds to` $\tilde{b} = \lfloor b/2^S \rfloor$
2 Remove last S bits from b, denote $\tilde{b}$
`// Regular comparison function,` $C \in \{0, 1\}$
3 $C =$ `Compare`$(\tilde{a}, \tilde{b})$
Output: C

We showcase the idea in the following example:

Example 4.6: *Suppose we have two points X and $Y \in \mathbb{R}^n$, and would like to assign some other points Z_i to one of two groups depending on whether they are closer to X or Y. We would do this by computing the distance of each point Z_i to both points X and Y and comparing the distances to find the smaller one. Suppose all involved numbers are rationals with a precision of 10 bits (i.e., scaled by 2^{10} and rounded to the nearest integer for encoding). Then setting $S = 5$ means that the comparison may be wrong if the* scaled *values are at most 2^5 apart – in other words, if the difference in the two distances between the underlying rational number Z_i and the two points X and Y is at most $\frac{2^5}{2^{10}} = 2^{-5} = 0.03125$, so X and Y are almost equally far from Z_i. This means that if we were comparing exactly, a slightly changed value for Z_i would assign it to the other group. In a stable algorithm (in the sense that small input perturbations do not significantly change the outcome), a wrong assignment of Z_i through the approximate comparison should thus not make a difference for the final result.* ———

We propose to pick an initial S and decrease it over the course of the algorithm, so that accuracy increases as we near the end. We will see the impact of the approximate comparison on the accuracy of the K-Means-Algorithm in Section 5.4. Regarding runtime, we measured the times for the comparison and approximate comparison functions with 35 bits total, and $S = 5$ bits deleted for approximate comparison. This yielded a drop in average (over 1000 runs each) runtime from 3.24 seconds for the regular comparison to 1.51 seconds for the approximate comparison. We see that this does make a big difference and may be of interest for computations involving many comparisons.

4.3.3 Length Management

Recall that by default, each addition and each multiplication increase the bitlength: Addition increases it by 1, whereas multiplication results in a bitlength that is the sum of the two input lengths. When performing several multiplications consecutively, this can easily lead to enormous bitlengths. However, in a scenario where the size of the values can be estimated, there is a way around this.
To see the validity of such an assumption, consider the scenario of machine learning as a service, where the person working on the encrypted data is the person who has the

algorithm for building the model. Here, it is a reasonable assumption that some factors of the model are known, e.g. from experience. For example, in the data set we will work with in Section 5.3, the value w_0 always takes some value near 10000 no matter what subset of test subjects we choose – thus, when computing on encrypted data, we might utilize this knowledge about our algorithm.
In such cases, the service provider who is doing the computations can put a bound on the lengths (i.e., he is certain that some value will not be larger in absolute value than 2^q for some q). When this is the case, we can reduce the bitlength of the encrypted values to this size $q + 1$ by discarding the excess bits: In Two's Complement, we can delete the most significant bits (which will all be 0 for a positive and 1 for a negative number) until we reach the desired length, whereas for Sign-Magnitude we discard the bits following the MSB (which will all be 0). To save even more computation time, we integrated this into our multiplication routine for Section 5.3, such that we not only save space, but also effort because we only compute until we reach the bound in each step.
This shortening operation can be viewed as the inversion of the sign extension introduced in Definition 3.1 and makes the entire algorithm significantly faster (see Section 5.3), as we have reduced the quadratic growth of the bitlength in multiplication.

4.4 CONCLUSION

In conclusion, we have developed two different ways of incorporating rational numbers into our FHE computations. The Fractional Encoding allows division of encrypted numbers, but has performance issues in practice. Our scaling procedure instead allows us to handle rationals as though they were integers, merely adding a new component which keeps the precision constant to the multiplication prodedure. This reduces the bitlength of the result while also hiding the function that was applied from the decrypting party. Additionally, it increases usability because the computing party does not need to keep track of the power of the scaling factor associated with each ciphertext.
In addition, we have showcased some other improvements to speed up the comparison and multiplication subroutines when certain assumptions hold true. The next chapter will present the impact of most of these improvements on the runtimes of real-world applications from the field of Machine Learning: Section 5.4.3 uses Fractional Encoding, Section 5.4.5 examines the impact of the approximate comparison, Section 5.3 shows the impact of length management, and the entire chapter (except Section 5.4.3) uses scaling with constant precision.

Chapter 5

APPLICATION TO MACHINE LEARNING

The previous chapters have dealt with encodings for FHE computations, which comprised the green box in Figure 1.1 in Section 1.3. In this chapter, we move on to the red box – namely applying the results of the previous chapters to algorithms from the field of Machine Learning, adapting the algorithms as necessary to improve performance or make them executable under FHE at all.

To this end, we first cover some preliminaries like the required background on Machine Learning and our runtime specifications in **Section 5.1**.

We then examine our first Machine Learning algorithm, the Linear Means Classifier, in **Section 5.2**. In this section, we assume that the model has already been trained, and we predict encrypted new unknown cases with this model. The results will showcase the impact of our Hybrid Encoding from Section 3.5.

We then turn to the other task in supervised Machine Learning, namely training the model, which we do for the Perceptron (a simple Neural Network) in **Section 5.3**. This will show the importance of the bounding procedure from Section 4.3.3 and again the improvement due to our Hybrid Encoding.

Lastly, we will move to the area of unsupervised learning by executing the K-Means-Algorithm, a clustering algorithm, on encrypted data in **Section 5.4**. We attempt to use the Fractional Encoding from Section 4.1, but we will see that our concerns from Section 4.1.3 about this encoding were indeed valid. We thus opt to change the underlying K-Means-Algorithm instead to avoid division, resulting in an FHE-friendly algorithm that achieves the same task with similar accuracy as the original one.

This chapter is largely taken from [JA16] and [JA18].

5.1 PRELIMINARIES

In this section, we will discuss some preliminaries: First, we will cover the basics of Machine Learning, including related work concerning Machine Learning on encrypted data. We then also present our implementation specifications, which were used for all the runtimes given in this work.

5.1.1 Introduction to Machine Learning

Machine Learning is a field of research that focuses on extracting information from datasets. If the dataset is very large, it is also often referred to as *Big Data* or *Data Mining*. There are countless algorithms in Machine Learning with inputs ranging from numeric over categorical to text-based. The applications today seem endless: We have the first self-driving cars, which have learned to do this via Neural Networks, we have smartphone keyboards that predict the next word based on your individual writing style, researchers are working on algorithms that can predict illness from a set of measured attributes or even a persons genome, and many more. However, many of these application scenarios involve sensitive data – people do not feel safe sending e.g. their medical data to a service provider, because they either do not trust the provider or are worried about a data breach even if they do trust the provider. This has lead to Machine Learning being a popular topic in the context of privacy-preserving computations in general, and Fully Homomorphic Encryption in particular.
Generally, Machine Learning can be divided into two categories: *supervised* and *unsupervised learning*.

5.1.1.1 *Supervised Learning*

In supervised learning, there is a dataset consisting of inputs and the correct outputs (which can be numerical values or classes into which the data is split) for these inputs. The goal is to build a model that correctly assigns these inputs to the outputs. This model can then be used to compute the outputs for new inputs, for which we do not know the correct answers.
Thus, algorithms from supervised learning consist of two parts: In the *training phase*, we build the model from the set of data with known outputs. The details of the method for deriving the model from the data are of course specific to the concrete Machine Learning algorithm we are using. We will implement this phase on encrypted data in Section 5.3 for the Perceptron, which is the earliest Neural Network.
Once we have a model, we move on to the *deployment phase*. Here, we feed new data points into the algorithm and obtain predictions for the outcome. We show how to perform this phase on encrypted data for the Linear Means Classifier, which classifies data via weighted sums, in Section 5.2.
Note that we have not mentioned the *testing phase* here: The concept is that a few data entries from the training set are put aside before training and are then used to measure the performance of the model once it has been built by comparing the predicted to the actual known outcomes. This phase is technically located between training and deployment phase, but for our high-level view we will simply consider it a part of the training phase.

5.1.1.2 *Unsupervised Learning*

In unsupervised learning, the situation is slightly different: There are no "correct" labels provided, so there is no training set – instead, the algorithm attempts to find some structure in the data on its own. An example of this is the clustering problem, which we will solve on encrypted data in Section 5.4, where the algorithm assigns the data entries to different clusters. Of course, the algorithms are not entirely automatic: In the clustering example, for many algorithms we need to specify how many clusters we expect, and the

algorithm must also have some kind of cost function, i.e., a way to compare two solutions to determine which is better. For many applications, this cost function is distance based (e.g., the average distance between two points in a cluster), but there exist many different application-specific cost functions.
Note that when developing new unsupervised algorithms, there may very well be a labeled testing set – these are usually artificially generated data sets for benchmarking purposes that allow the comparison of different algorithms on the same dataset, e.g. by comparing the percentage of correctly labeled data points. However, this takes place in the development phase of the algorithm and is not part of the Machine Learning process once the algorithm has been established.

5.1.1.3 *Related Work*

While some of this related work has already been covered in Section 1.1, we still choose to give a comprehensive overview at this point, repeating the former as necessary.
Machine Learning as an application for Fully Homomorphic Encryption was first proposed in [GLN12], and since then it has been a popular area of research. There are many areas of Machine Learning that have been studied in the context of FHE, and we give a brief overview of the most popular ones.
The first of these areas that many works have focused on is (Deep) Neural Networks, where input nodes are connected to output nodes through (sometimes numerous) intermediate layers. Our publication [JA16] implements the Perceptron [Ros57], which is a Neural Network without any intermediate layers and is thus a building block for the more complicated versions. In [BMMP17], the bootstrapping procedure of the underlying encryption scheme is modified to accommodate a discretized Neural Network, whereas [GDL$^+$16] and [CdWM$^+$17] adapt the different layers of a Deep Neural Network through polynomial approximations of the functions in question. Works like [PAH$^+$17] and [JVC18] rely on interactive solutions from the realm of Multiparty Computation, often in combination with FHE building blocks.
First suggested in [BLN14], there has been a recent surge of papers dealing with the task of logistic regression on encrypted data. This is a widely used algorithm in Machine Learning, but it is non-trivial to implement on encrypted data because of the non-polynomial Sigmoid function $s(z) := \frac{1}{1+e^{-z}}$ involved in the computation. In [KSW$^+$18], this problem is tackled by using a least-squares approximation for the Sigmoid function, whereas [KSK$^+$18] and [BV18] use a local polynomial approximation, and [BCG$^+$17] uses multiparty computation and a Fourier approximation of the Sigmoid Function. In [CGH$^+$18], the problem is solved in a manner very specific to the underlying FHE library HElib [LIBd], and the user must solve a linear system of equations to obtain the result of the computation after decrypting.
Another popular area of research is (Linear) Regression or Hyperplane Decision, where a hyperplane is fitted and data points are classified according to which side of the hyperplane they lie on. Publications concerned with this task are [GLN12], [BPTG15], [LKS16] and [EAH17].
Other algorithm classes that have been considered include decision trees and random forests in [WFNL16], [BPTG15] and [AEH15], Support Vector Machines in [BSS$^+$17], and Naive Bayes Classification in [AEH15] and [BPTG15], though many of these solutions rely on Multiparty Computation and thus interaction between the data owner and the computing party during the computation.

For the area of unsupervised learning, our publication [JA18], which implements the K-Means-Algorithm [M$^+$67], is to our knowledge the only work concerned with this research area of unsupervised Machine Learning on encrypted data via FHE. The K-Means-Algorithm has been a subject of interest in the context of privacy-preserving computations for some time, but to our knowledge all previous works like [BO07], [JW05], [JPWU10], [LJY$^+$15] and [XHY$^+$17] require interaction between several parties, e.g. via Multiparty Computation (MPC). For a more comprehensive overview of the K-Means-Algorithm in the context of MPC, we refer the reader to [MB12]. While this interactivity may certainly be a feasible requirement in many situations, and indeed MPC is likely to be faster than FHE in these cases, we feel that there is nonetheless a need for a non-interactive solution as we present it: FHE reduces the computational load of the user to zero, and it also allows the computing party to keep the function secret (if the FHE scheme has circuit privacy, see Definition 1.7, which all current schemes do). Also, many of these interactive solutions rely on a vertical (in [VC03]) or horizontal (in [JKM05]) partitioning of the data between several users for security. In contrast, FHE allows a non-interactive setting with a single database owner who wishes to outsource the computation.

5.1.2 Implementation Specifications

All runtimes in this thesis, including the ones from previous chapters, were measured on a virtual machine with 20 GB of RAM and 4 virtual cores, running Ubuntu 16.04 on an Intel i7-3770 processor with 3.4 GHz. We used the TFHE library [LIBf], as it is currently the fastest one available for $\{0,1\}$-plaintext spaces. We also note that the documentation for this library states *"Since the running time per gate seems to be the bottleneck of fully homomorphic encryption, an optimal circuit for TFHE is most likely a circuit with the smallest possible number of gates, and to a lesser extent, the possibility to evaluate them in parallel."*. This means that we do not need to minimize depth in our computations, as the total number of gates is the relevant metric. We ran the library without the SPQLIOS_FMA-option that enables faster Fourier transforms, which are used in the encryption library, as our processor did not support this (runtimes might be faster when using this option). Note that the TFHE library currently only supports single thread computations, so only one of the four cores was actually used.

5.2 THE LINEAR MEANS CLASSIFIER

We start with the most simple case where we already have a model, which may have been trained on encrypted or unencrypted data, and the task is to apply this model to encrypted data. This could be the case in the context of Machine Learning as a service, where a provider owns the algorithm and offers to perform predictions on user data for a fee. It could also be the case if the algorithm and the data belong to the user, who merely wants to outsource the computation. By encrypting the data, the user can protect his data and still use this service. For maximum generality, we assume that the model coefficients are encrypted as well – however, if the model belongs to the service provider, the coefficients would likely not be encrypted.

5.2.1 Algorithm Details

As already implied by its name, the Linear Means classifier classifies data – i.e., it assigns data entries to one of several possible classes. Like [GLN12], we consider the case where there are two classes, which are determined by the sign of the score function. This score function is a polynomial of degree 2. More concretely, the model consists of a vector $w = (w_1, \ldots, w_l)$ and a threshold constant τ, and the data to be classified is a l-dimensional real-valued vector $x = (x_1, \ldots, x_l)$. The l traits of the data are also known as *features*. The score function is then computed as

$$\langle w, x \rangle + \tau = w_1 x_1 + w_2 x_2 + \cdots + w_l x_l + \tau, \tag{5.1}$$

and the sign of the result determines which class the data instance belongs to[1]. Thus, we now need to compute the score function for given encrypted values for w and τ. The code for the unencrypted Linear Means Classifier can be seen in Algorithm 11.

Algorithm 11: `Linear Means Classification`

Input: The weight vector $w = (w_1, \ldots, w_l)$ and the threshold constant τ
Input: The input vector $x = (x_1, \ldots, x_l)$
Compute $sum = \langle w, x \rangle + \tau = w_1 x_1 + w_2 x_2 + \cdots + w_l x_l + \tau$
if $sum < 0$ **then**
  $res = 1$
else
  $res = 0$
end
Output: res

When the data is encrypted bitwise as in our case, we can easily implement the `if`-branch by taking the MSB of the result, which is 1 when the number is negative, and 0 when it is non-negative. The pseudocode using the subroutines `Add` and `Mult` for encrypted addition and multiplication can be seen in Algorithm 12.

[1] We will see that this score function is closely related to the classification function of the Perceptron from Section 5.3, where the focus will be on determining w and τ. Here, we instead see these values as given and use them to compute predictions on input data.

Algorithm 12: `Encrypted Linear Means Classification`

Input: The bitwise encrypted weight vector $w = (w_1, \ldots, w_l)$
Input: The bitwise encrypted threshold constant τ
Input: The bitwise encrypted input vector $x = (x_1, \ldots, x_l)$
$sum = \tau$
for $i = 1$ *to* l **do**
  $temp = \texttt{Mult}(w[i], x[i])$
  $sum = \texttt{Add}(temp, sum)$
end
`// Take the sign of the sum:`
$res = MSB(sum)$
Output: res

We see that this is easily implemented through routines from the previous chapters, and we present the performance in the following.

5.2.2 Performance

Using the Linear Means Classifier, we now examine the effects of using different encodings in the unbounded case (i.e., when the product of two n-bit numbers has length up to $2n$). To this end, we measured the runtime of evaluating the score function for inputs of bitlength 30 for different numbers l of features. Note that of course, the number l of features is known to the computing party, as it is revealed by the length of the encrypted data vector. The results can be seen in Figure 5.1.

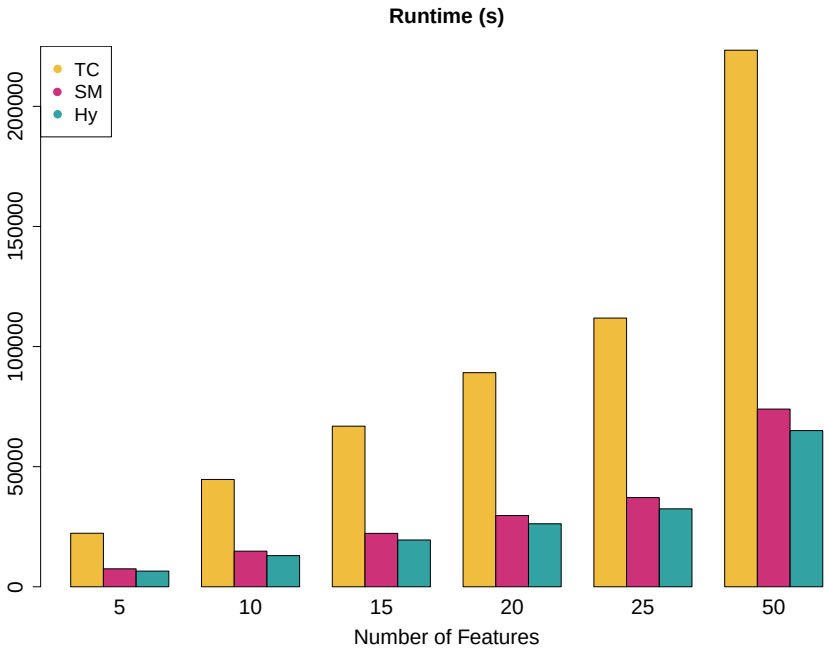

Figure 5.1: Runtime for the Linear Means Classifier on encrypted data for different numbers of features with encoding length 30.

We can see that the new Hybrid Encoding outperforms the other two encodings – concretely, its runtime on average is only 29% that of the Two's Complement encoding, and 88% that of the Sign-Magnitude encoding.

5.2.3 Conclusion

We have seen in this Section that our new Hybrid Encoding can improve runtime by a significant factor, and also that simple tasks from the realm of Machine Learning can be performed on encrypted data without complications. Of course, the majority of the algorithms that are used in practice are significantly more complicated than the Linear Means Classifier. In this section, we have used an already trained model to predict new occurrences by evaluating a multivariate polynomial of degree 2 (see Equation 5.1 on page 111) on encrypted data. The next section will deal with the model training – that is, the more complicated task of deriving the weights w and the threshold constant τ in the first place.

5.3 THE PERCEPTRON

We now move on to the task of obtaining a model, e.g. the coefficients for the Linear Means Classifier above, in the training phase. We use the Perceptron for this – it is a simple Neural Network that will be explained in Section 5.3.1. In our scenario, the training data will be encrypted, which would be feasible in e.g. the following scenario: Suppose one party has developed a new prediction algorithm, and would like to offer it as a service. However, the accuracy of any predictive algorithm generally improves if the training data comes from the same population as the new data later on, so for the best predictions, the model must also be trained on the user data. For example, the population in our Perceptron dataset is a real-world dataset relating Diabetes to other factors in a population of female Native Americans from the Pima tribe. It is not hard to imagine that if the model were trained on the general population (or, say, European males), the predictive accuracy on Pima Indian females would not be as high as when the model is also trained on this group. However, if we train the model on the user data, the same privacy concerns as during the classification phase are relevant – that is, the user would like to protect his sensitive data through encryption. Thus, this section focuses on training a model (the Perceptron) on encrypted training data. Once this is done, the party with the algorithm can either send the algorithm to the user, who can decrypt it and use it on his own in unencrypted form, or the model can be kept in encrypted form by the algorithm developer and be used to predict future encrypted instances from the user as a (paid) service.

This section will also illustrate the impact of the length management procedure from Section 4.3.3, as we feel that it fits well into this scenario: It is likely that the algorithm developer has some information about the ranges of the coefficients of his model, as perhaps they do not vary wildly across different populations. In this case, he can use the length management procedure to keep the bitlengths in the neccessary ranges (with enough room for variations in the values) instead of having them grow quadratically with each multiplication.

5.3.1 Algorithm Details

The Perceptron is a simple Neural Network and works by computing a weighted sum

$$\sum_{k=1}^{K} w_k \cdot x_{i,k}$$

of K input traits $x_{i,k}$, which are usually rational numbers, for each subject x_i and then classifying into one of two classes depending on whether this weighted sum is above a certain threshold or not. In the training phase, the weights w_k are adjusted if the computed classification does not match the known classification $c_i \in \{0, 1\}$ of the training instance x_i. The learning rate η determines by how much the weights change through each such mismatch. A larger η means bigger changes and less stability, whereas for a smaller value, more rounds may be needed, but the weights are more likely to converge. After training, the model can be used to classify inputs with no known classification by again computing the weighted sum and comparing to the threshold. The model consists of the weights, and the threshold can either be predetermined or flexible (and thus part of the model being computed). We will work with the latter approach and for notation reasons include a dummy trait that is always -1, which enables us to compare the scalar product to 0: Again denoting the threshold as τ, we have

$$\sum_{k=1}^{K} w_k \cdot x_k > \tau \Leftrightarrow \sum_{k=0}^{K} w_k \cdot x_k > 0$$

for $x_0 = -1$ and $w_0 = \tau$.
Note that this is a binary classifier, i.e., it only works with two classes, but the more complex case of several classes can easily be built by running the Perceptron several times for different classes or by having more than one output (i.e., computing several sums and having the targets not be bits, but bitstrings). The exact workings of the Perceptron are presented in Algorithm 13, and further implementation details can be found in the following.

Algorithm 13: `The Perceptron`

`Training():`

`// m subjects with K traits`

Input: Training Data $(x_{i,k}, c_i)$, $i = 1, \ldots, m$, $k = 1, \ldots, K$, where $x_{i,k}$ are the features and c_i is the class

Input: Learning Rate η

Input: Iteration Number T

Write inputs as X, a $m \times (K+1)$-matrix with first column -1, followed by the inputs x_i row-wise, and m-dimensional target vector $\vec{c}$ with entries c_i.

for $k = 0$ *to* K **do**

`// Initialize weights to small random numbers`

$w_k \leftarrow \mathcal{R}_\epsilon$

end

for T *iterations* **do**

for $i = 1$ *to* m **do**

`// Compute prediction for current subject`

if $\sum_{k=0}^{K} w_k \cdot x_{i,k} > 0$ **then**

Set $y = 1$

else

Set $y = 0$

end

`// If prediction equals class, the weights don't change. If class is 0 but prediction is 1, a fraction of that feature value is subtracted from the weight. If class is 1 but prediction is 0, a fraction of that feature value is added to the weight.`

for $k = 0$ *to* K **do**

$w_k \leftarrow w_k + \eta \cdot (c_i - y) \cdot x_{i,k}$

end

end

end

Output: The weights w_k, $k = 0$ to K

`Classification():`

Input: $w_0, \ldots, w_K$ from Training Phase

Input: Vector $x = (x_1, \ldots, x_K)$ to be classified

Set $x_0 = -1$

if $\sum_{k=0}^{K} w_k \cdot x_k > 0$ **then**

Set $y = 1$

else

Set $y = 0$

end

Output: The classification y

Note that to compute the Perceptron on bitwise encrypted data, we merely replace additions and multiplications with the appropriate routine on encrypted data. There is, however, one computation we pay special attention to: To reduce runtime, in our implementation we chose to rewrite the term $\eta \cdot (c_i - y) \cdot x_{i,k}$ in Line 13 as $\eta \cdot c_i \cdot x_{i,k} - \eta \cdot x_{i,k} \cdot y$. This way, we can precompute the values $\eta \cdot c_i \cdot x_{i,k}$ and $-\eta \cdot x_{i,k}$ for all $i = 0, \ldots, m$ and all $k = 0, \ldots, K$, as these values do not change from round to round.
The multiplication for these terms is easier as well: We first compute the terms $\eta \cdot x_{i,k}$ and $-\eta \cdot x_{i,k}$, where we can use multiplication with a constant (denoted `ConstMult`) because η is not encrypted (see Section 3.2.2.1). To get the first precomputation term $\eta \cdot c_i \cdot x_{i,k}$, we must multiply $\eta \cdot x_{i,k}$ by c_i. However, it holds that $c_i \in \{0, 1\}$ – thus, we do not need a complicated multiplication procedure involving multiple addition subroutines, but can instead multiply each bit of $\eta \cdot x_{i,k}$ by c_i – this results in $\eta \cdot x_{i,k}$ if $c_i = 1$, and in 0 otherwise. The same holds in the main loop when we compute $(\eta \cdot x_{i,k}) \cdot y$: As $y \in \{0, 1\}$, we only need to do a simple bit-for-bit multiplication, which is much faster than regular multiplication. Denoting this bit-for-bit multiplication as `OneBitMult` (see Algorithm 14), the training phase on encrypted data with precomputation can be seen in Algorithm 15. The 0-comparison is the easy comparison from Section 4.3.1, and the significant changes compared to the algorithm on unencrypted data are shaded.

Algorithm 14: `OneBitMult`(c, a)

Input: An encrypted bit c
Input: A bitwise encrypted number $a = a_n a_{n-1} \ldots a_1 a_0$
1 **for** $i = 0$ *to* n **do**
2 $\quad res_i = c \cdot a_i$
3 **end**
Output: res

Algorithm 15: `Training the Perceptron on encrypted data`

`Precomputation ():`

Input: Training Data $(x_{i,k}, c_i)$ (encrypted)
Input: Learning Rate η (not encrypted)
1 **for** $i = 1$ *to* m **do**
2 **for** $k = 0$ *to* K **do**
 // $temp = \eta \cdot x_{i,k}$
3 $temp = \texttt{ConstMult}(\eta, x_{i,k})$
 // $p1_{i,k} = c_i \cdot \eta \cdot x_{i,k}$
4 $p1_{i,k} = \texttt{OneBitMult}(c_i, temp)$
 // $p2_{i,k} = -\eta \cdot x_{i,k}$
5 $p2_{i,k} = \texttt{ConstMult}((-\eta), x_{i,k})$
6 **end**
7 **end**
Output: $p1_{i,k}$ and $p2_{i,k}$ for $i = 1$ to m and $k = 0$ to K (encrypted)

`Training():`

// m `subjects with` K `traits (encrypted bitwise)`
Input: Training Data $(x_{i,k}, c_i)$ (encrypted)
Input: Outputs of precomputation $p1_{i,k}$ and $p2_{i,k}$ (encrypted)
Input: Iteration Number T (not encrypted)
8 **for** $k = 0$ *to* K **do**
 // `Initialize weights to small random numbers -- this can be done in unencrypted form, then the multiplications in the first iteration of the main loop will be ConstMult instead of Mult. After that, the weights are encrypted values.`
9 $w_k \leftarrow \mathcal{R}_\epsilon$
10 **end**
11 **for** T *iterations* **do**
12 **for** $i = 1$ *to* m **do**
 // `Compute the weighted sum`
13 $sum = -w_0$
14 **for** $k = 1$ *to* K **do**
15 $sum = \texttt{Add}(sum, \texttt{Mult}(w_k, x_{i,k}))$
16 **end**
 // `The prediction is the flipped sign of the sum`
17 $y = \texttt{Sign}(sum) \oplus 1$
 // `Update the weights`
18 **for** $k = 0$ *to* K **do**
19 $temp = \texttt{Add}(p1_{i,k}, \texttt{OneBitMult}(y, p2_{i,k}))$
20 $w_k = \texttt{Add}(w_k, temp)$
21 **end**
22 **end**
23 **end**
Output: The bitwise encrypted weights w_k, $k = 0$ to K

5.3.1.1 *Dataset and Parameters*

We will now discuss the parameters and the dataset used in our actual implementation of the Perceptron. To test our implementation, we used the Pima Indian Dataset [Pim]. This dataset is a subset of a larger dataset collected by the National Institute of Diabetes and Digestive and Kidney Diseasesis starting in 1965, and was first used in [SED+88]. The dataset consists of entries for 768 females of at least 21 years of age, and measurements were taken at various points in time, but only one measurement per subject was included. The entries consist of 8 different traits and are classified into "developed diabetes within 5 years" or "did not develop diabetes within 5 years". Since the weights for two attributes did not seem to converge at all, we reduced the number of traits down to $K = 6$:

1. Number of times pregnant
2. Plasma glucose concentration at 2 hours in an oral glucose tolerance test
3. Diastolic blood pressure (mm\Hg)
4. Body mass index $\frac{\text{weight in kg}}{(\text{height in m})^2}$
5. Diabetes pedigree function
6. Age (years)

We set the learning rate to $\eta = 0.125$ and reserved some subjects for the testing phase (i.e., use the weights obtained from the model to classify entries with known class which were not involved in the generation of the weights and see how many are correct). Note that the testing phase is not carried out bitwise and does not really belong to our encrypted model, but we performed it to see how different precision values influenced the accuracy of our derived model. We used different precision values and bounds on the length, and these computations were done in unencrypted form, encoded bitwise, as they were only to determine satisfactory parameter values. As it turns out, 20 bits with 6 bits for precision is not enough (i.e., there was an overflow and the results were wrong), whereas both bitlengths 25 (precision 10) and 30 (precision 15) yielded satisfactory results in that the values were correct for our computations. From previous experiments, we knew that w_0 (i.e., the weight multiplied with $x_0 = -1$) always converged to a number around 10000, so we initialized w_0 as $10000 + r$ where r is a small random number. As already mentioned above, we feel that this is a feasible scenario because the computing party has some knowledge about the model that is being trained on the user's encrypted data. Note that we could live without this assumption, but we would need more rounds, as the weight only changes by a small amount in each weight update and it would take many rounds to reach a value as big as 10000.

5.3.2 Performance

We now present the runtimes for the three parameter sets (20 bits with 6 bits of precision, 25 bits with precision 10, and 30 bits with precision 15) for the case with and without length management. Note that as for all our runtime measurements, these numbers heavily depend on the encryption library we used: In [LIBf], which we used, all gates take roughly the same time, whereas e.g. in [LIBd], bit multiplication takes much longer than bit

addition. Thus, these runtimes give an indication of the performance gains we can achieve, but the concrete times are specific to the underlying library. The results can bee seen in Figure 5.2.

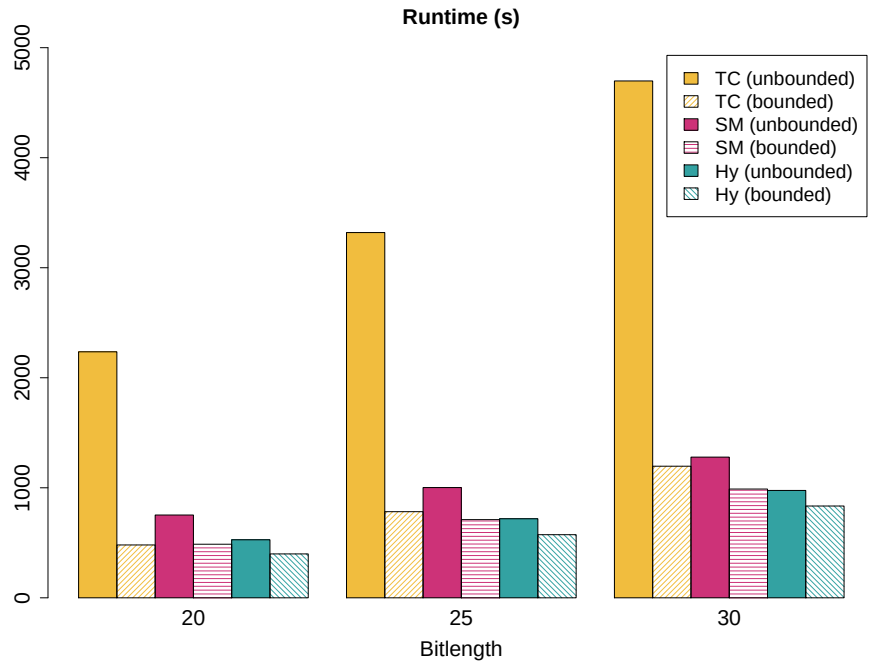

Figure 5.2: Runtime for one round of the Perceptron on encrypted data for one subject for encoding lengths $20, 25$ and 30, with and without length management.

The times refer to one round for one subject (averaged over 5 runs). We can see that the length management procedure makes the computations significantly faster – in the unbounded scenario, the times would increase further with each round and subject due to increasing bitlengths, making the impact of the length management procedure even more profound than Figure 5.2 suggests. For the bounded cases, we can expect runtimes per round to stay constant over the course of the algorithm. The length management has the biggest impact on the Two's Complement encoding, which is to be expected: The multiplication requires sign extension to double the length (see Section 3.2.2), so that longer input lengths slow this encoding down much more than the other two. However, the bounding procedure decreases the runtime for all three encodings, leading to significant efficiency gains.

We can also see from Figure 5.2 that the Hybrid Encoding again outperforms the other two encodings in all parameter settings: If there is no length management, the Hybrid Encoding has on average only 22% the runtime of Two's Complement encoding, and 73% that of the Sign-Magnitude encoding. In the bounded case, it has 75% the runtime of Two's Complement, and 82% that of Sign-Magnitude. The ratio of the best setting (bounded Hybrid Encoding) and the worst (unbounded Two's Complement) is 0.176, so we can save over 82% runtime through our improvements.

5.3.3 Conclusion

In conclusion, we have shown how even a more complicated procedure like training a neural network can be accomplished on encrypted data. Furthermore, the choice of the encoding has a big impact, with our Hybrid Encoding outperforming the other two encodings in all parameter settings. If we can estimate the size of the involved values in an iterative computation, we can reduce runtime even further by keeping the bitlength constant and thus eliminating the bitlength expansion during addition and multiplication.
We have now covered both components (training and prediction) of supervised learning – the next section will deal with the other main pillar of Machine Learning, namely unsupervised learning.

5.4 CLUSTERING ON ENCRYPTED DATA

The problem we tackle now is that of clustering: The input consists of some data points, and the aim is to group entries together that are similar in some way. The number of clusters may be a parameter that the user enters, or it may be automatically selected by the algorithm. Clustering has numerous applications like genome sequence analysis, market research, medical imaging or social network analysis, to name just a few. Many of these applications inherently involve sensitive data – making a privacy-preserving evaluation with FHE even more interesting.
The clustering algorithm we choose is the K-Means-Algorithm, which is an established solution to the problem. We first use the Fractional Encoding from Section 4.1 to implement the K-Means-Algorithm as it is (though we change the distance metric after showing that the choice is arbitrary in the first place), but we will see that our concerns about this encoding from earlier are valid, as the runtime is prohibitively large, and parameters to ensure correctness are difficult to set. Thus, we instead opt to modify the underlying algorithm to avoid the division that made the Fractional Encoding necessary. We show that this modified algorithm performs comparably to the original K-Means-Algorithm in terms of accuracy while being significantly faster, and also examine how the use of the approximate comparison from Section 4.3.2 affects the runtime.

5.4.1 Algorithm Details

The K-Means-Algorithm is one of the most well-known clustering algorithms in unsupervised learning. Published in [M+67], it is considered an important benchmark algorithm and is frequently the subject of current research to this day.
The K-Means-Algorithm takes as input the data

$$X = \{x_1, \ldots, x_m\},\ x_i = (x_i^{(1)}, \ldots, x_i^{(\ell)}) \in \mathbb{R}^\ell,$$

and a number K of clusters to be used. It begins by choosing either K random values in the data range or K randomly chosen data entries as so-called **cluster centroids** μ_k. We will use the latter approach. Then, in a step called `Cluster Assignment`, it computes for each data entry x_i which cluster centroid μ_k is nearest regarding Euclidean distance

$$||x_i - \mu_k||_2 = \sqrt{\sum_j (x_i^{(j)} - \mu_k^{(j)})^2},$$

and assigns the data entry to that centroid (denoted $x_i \in \mu_k$). When this has been done for all data entries, the second step begins: During the `Move Centroids` step, the cluster centroids are moved by setting each centroid as the average of all data entries that were assigned to it in the previous step:

$$\mu_k = \frac{\sum\limits_{x_i \in \mu_k} x_i}{|x_i \in \mu_k|}.$$

These two steps are repeated for a set number of times T or until the centroids do not change anymore. We use the first method – a visualization of the K-Means-Algorithm can be found in Figure 5.3.

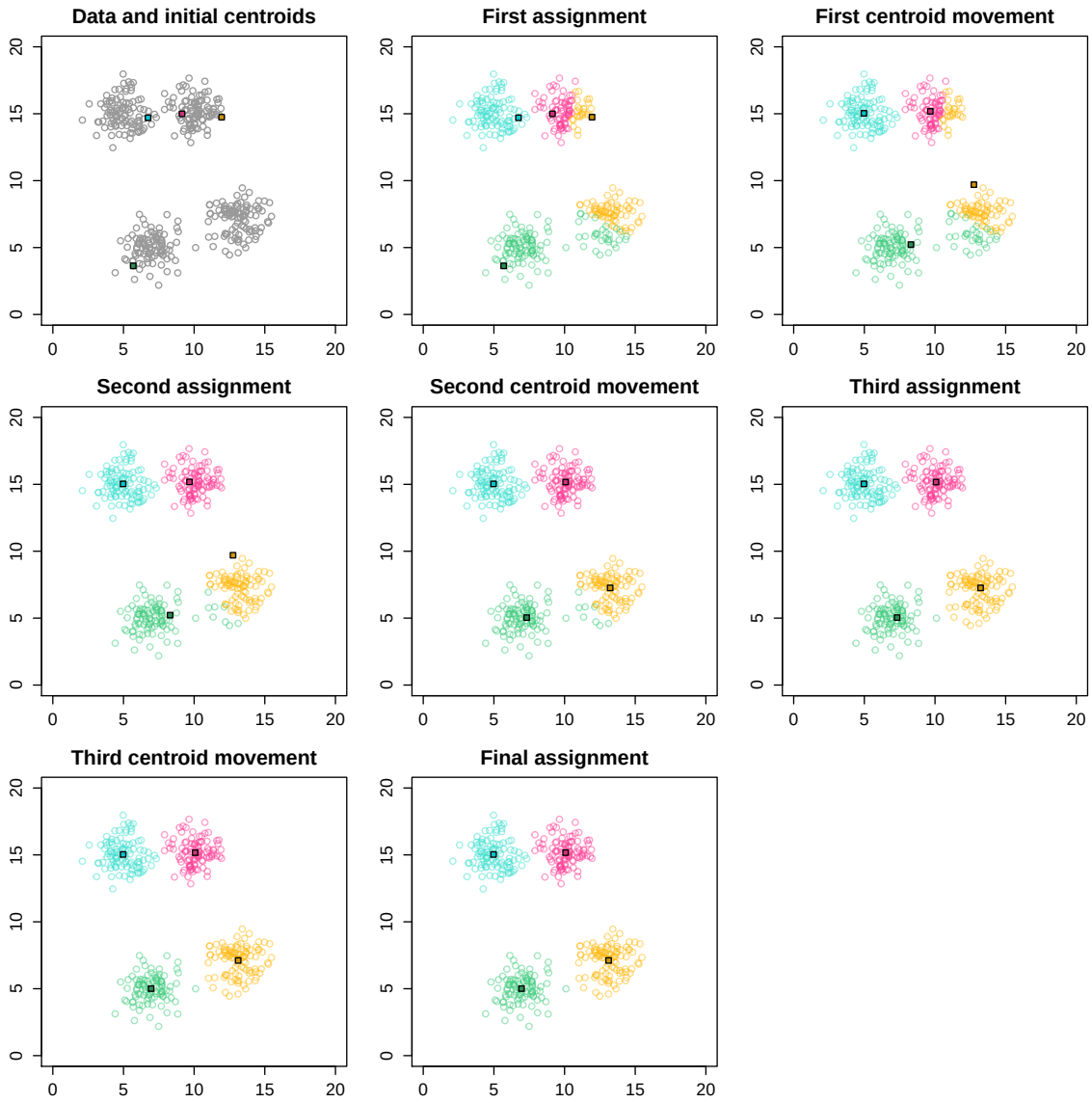

Figure 5.3: An illustration of the K-Means-Algorithm.

The output of the algorithm is the values of the centroids, or the cluster assignment for the data entries (which can easily be computed from the former). We opt for the first approach. The exact workings of the K-Means-Algorithm can be seen in Algorithm 16, where operations like addition and division are performed component-wise if applied to vectors. A list of the variables used in the algorithm can be found in Section 5.4.1.1.
Accuracy can either be measured in terms of correctly classified data entries, which assumes that the correct classification is known (there might not even exist a unique best solution), or via the so-called cost function, which measures the (average) distance of the data entries to their assigned cluster centroids. We opt for the first approach because our datasets are benchmarking sets for which the labels are indeed provided, and it allows better comparability between the different algorithm versions.

5.4.1.1 *List of Variables*

To aid the reader, we present a brief recap of the variables that we use in dealing with the K-Means-Algorithm:

- K: Number of clusters.
- μ_k: Cluster centroid k.
- m: Number of data points.
- $X = \{x_1, \ldots, x_m\}, x_i \in \mathbb{R}^\ell$: The dataset.
- ℓ: The dimension of the data.
- d_k: Denominator of centroid k in the average computation (i.e., the number of data entries assigned to that cluster: $d_k = |x_i \in \mu_k|$).
- T: Number of rounds to run the algorithm.
- Δ: A number to hold distances (later: a vector).
- A: The cluster assignment vector (m-dimensional), later a boolean matrix ($m \times K$).

5.4.1.2 *Datasets*

To evaluate performance, we use four datasets from the FCPS dataset [Ult05] to monitor performance:

- The Hepta dataset consists of 212 data points of 3 dimensions. There are 7 clearly defined clusters.
- The Lsun dataset is 2-dimensional with 400 entries and 3 classes. The clusters have different variances and sizes.
- The Tetra dataset is comprised of 400 entries in 3 dimensions. There are 4 clusters, which almost touch.
- The Wingnut dataset has only 2 clusters, which are side-by-side rectangles in 2-dimensional space. There are 1016 entries.

Algorithm 16: The K-Means-Algorithm

Input: Data set $X = \{x_1, \ldots, x_m\}$ // $x_i = (x_i^{(1)}, \ldots, x_i^{(\ell)}) \in \mathbb{R}^\ell$ for some ℓ
Input: Number of clusters K
Input: Number of iterations T
// Initialization (choose random starting centroids)
Randomly reorder X
Set centroids $\mu_k = x_k$ for $k = 1$ to K
// Keep track of centroid assignments
Generate m-dimensional vector A
// Keep track of denominators in average computation
Generate K-dimensional vector $d = (d_1, \ldots, d_K)$
for $j = 1$ *to* T **do**
  // Cluster Assignment
  for $i = 1$ *to* m **do**
    $\Delta = \infty$
    for $k = 1$ *to* K **do**
      $\tilde{\Delta} := ||x_i - \mu_k||_2 = \sqrt{\sum_j (x_i^{(j)} - \mu_k^{(j)})^2}$
      // Check if current cluster is closer than previous closest
      if $\tilde{\Delta} < \Delta$ **then**
        // If so, update Δ and assign data entry to current cluster
        $\Delta = \tilde{\Delta}$
        $A_i = k$
      end
    end
  end
  // Move Centroids, i.e., compute $\mu_k = \frac{\sum_{x_i \in \mu_k} x_i}{|x_i \in \mu_k|}$.
  for $k = 1$ *to* K **do**
    $\mu_k = 0$
    $d_k = 0$
  end
  for $i = 1$ *to* m **do**
    // Add the data entry to its assigned centroid
    $\mu_{A_i} = \mu_{A_i} + x_i$
    // Increase the appropriate denominator
    $d_{A_i} = d_{A_i} + 1$
  end
  for $k = 1$ *to* K **do**
    // Divide centroid by number of assigned data entries to get average
    $\mu_k = \mu_k / d_k$
  end
end
Output: $\{\mu_1, \ldots, \mu_K\}$

For accuracy measurements, each version of the algorithm was run 1000 times for number of iterations $T = 5, 10, ..., 45, 50$ on each dataset. For runtimes on encrypted data, we used the Lsun dataset.

5.4.1.3 *FHE Challenges*

Before we discuss the details of implementing the K-Means-Algorithm, we first address the two challenges that arise in the context of FHE computation of this algorithm and quickly explain how we solve them. The line numbers refer to Algorithm 16.

- The distance metric (Line 9, $\Delta(x_i, \mu_k) := ||x_i - \mu_k||_2 = \sqrt{\sum_j (x_i^{(j)} - \mu_k^{(j)})^2}$):
  To our knowledge, taking the square root of encrypted data has not been implemented yet, and discarding the square root and just using the sum of squared differences $\sum_j (x_i^{(j)} - \mu_k^{(j)})^2$ would lead to very large bitlengths and enormous runtimes. In Section 5.4.2, we will argue that the Euclidean norm is an arbitrary choice in this context and solve this problem by using the L_1-distance

  $$\Delta(x_i, \mu_k) = ||x_i - \mu_k||_1 := \sum_j (|x_i^{(j)} - \mu_k^{(j)}|)$$

  instead of the Euclidean distance.

- Division (Line 25, $\mu_k = \mu_k / d_k$) in computing the new centroid value as the average of the assigned data points:
  Recall that in FHE computations, division by an encrypted value is usually not possible (whereas division by an unencrypted value is no problem). In Section 5.4.3, we first use the Fractional Encoding from Section 4.1 to allow this division, and then propose a modified version of the K-Means-Algorithm in Section 5.4.4 that only needs division by a constant.

5.4.2 The Distance Metric

Traditionally, the distance measure used with the K-Means Algorithm is the Euclidean Distance

$$\Delta(x, y) = \ ||x - y||_2 := \sqrt{\sum_j (x^{(j)} - y^{(j)})^2},$$

also known as the L_2-Norm, as it is analytically smooth and thus reasonably well-behaved. However, in the context of K-Means Clustering, smoothness is irrelevant (as we will not be taking any derivatives), and we may look to other distance metrics. Concretely, we consider the L_1-Norm[2] (also known as the Manhattan-Norm)

$$\Delta(x, y) = ||x - y||_1 := \ \sum_j (|x^{(j)} - y^{(j)}|).$$

This has a considerable advantage over the Euclidean distance: Firstly, we do not need to take a square root, which to our knowledge has not yet been achieved on encrypted

[2] [AHK01] in fact argues that for high-dimensional spaces, the L_1-Norm is more meaningful than the Euclidean Norm.

data. Secondly, of course one could apply the standard trick and not take the root, working instead with the sum of squared distances – however, this would mean a considerable efficiency loss. To see this, first note that multiplying two numbers takes significantly longer than taking the absolute value. Also, recall that multiplying two numbers of equal bitlength results in a number of twice that bitlength. These much longer numbers then have to be summed up, and already the summation step is a bottleneck of the whole computation on encrypted data even when working with short numbers in the L_1 norm. The result of the summation is an input to the algorithm that finds the minimum (Algorithm 19 on page 130), which also takes a significant amount of time and would likely more than double in runtime if the input length doubled.
Taking the absolute value can easily be achieved by using the MSB as the conditional (recall that it is 1 if the number is negative and 0 if it is positive) and use a multiplexer gate applied to the value and its negative. The concrete algorithm can be seen in Algorithm 17.

Algorithm 17: `Absolute Value`

Input: Value $a = a_n \dots a_1 a_0$ in Two's Complement or Sign-Magnitude encoding

`// Set the conditional variable as the MSB`

1 $C = MSB(a) = a_n$

`// Apply the MUX gate`

2 $d = MUX(C, \texttt{Invert}(a), a)$

`// If` $C = 1$`, i.e., a is negative,` $d = -a$`. If` $C = 0$`, i.e., a is positive,` $d = a$`. So` $d = |a|$`.`

Output: d

Thus, using the L_1-Norm is not only justified by the arbitrariness of the Euclidean Norm, but is also much more efficient on encrypted data. To compare the clustering accuracy, recall that the datasets we are working with are benchmarking sets designed specifically to compare the performance of different clustering algorithms. For this purpose, the data points have labels that assign them to the "correct" clusters, so that we can compare two algorithms by comparing the accuracy, i.e., the percentage of correctly labeled data points[3]. Concretely, we calculated the percentage of wrongly labeled data points for 1000 runs (i.e., for each run choosing random starting centroids and then running both algorithms from these same starting centroids), for both versions of the distance metric. We then plotted histograms of the difference (in percent mislabeled) between the L_1-norm and the L_2-norm for each run. Thus, a value of 0.5 means that starting from the same centroids, the L_1 norm version misclassified 0.5% more data entries than the L_2-version, and -2 means that the L_1 version misclassified 2% less data entries than the L_2-version. Each subplot corresponds to one of the four datasets we used. The comparison of the clustering accuracy can be seen in Figure 5.4.

[3]Recall that in unsupervised learning, the data generally has no correct outputs provided, and the notion of "correct" is usually not even well-defined for clustering problems. This special case of a benchmarking set is an exception that allows us to compare the performance between two different algorithms.

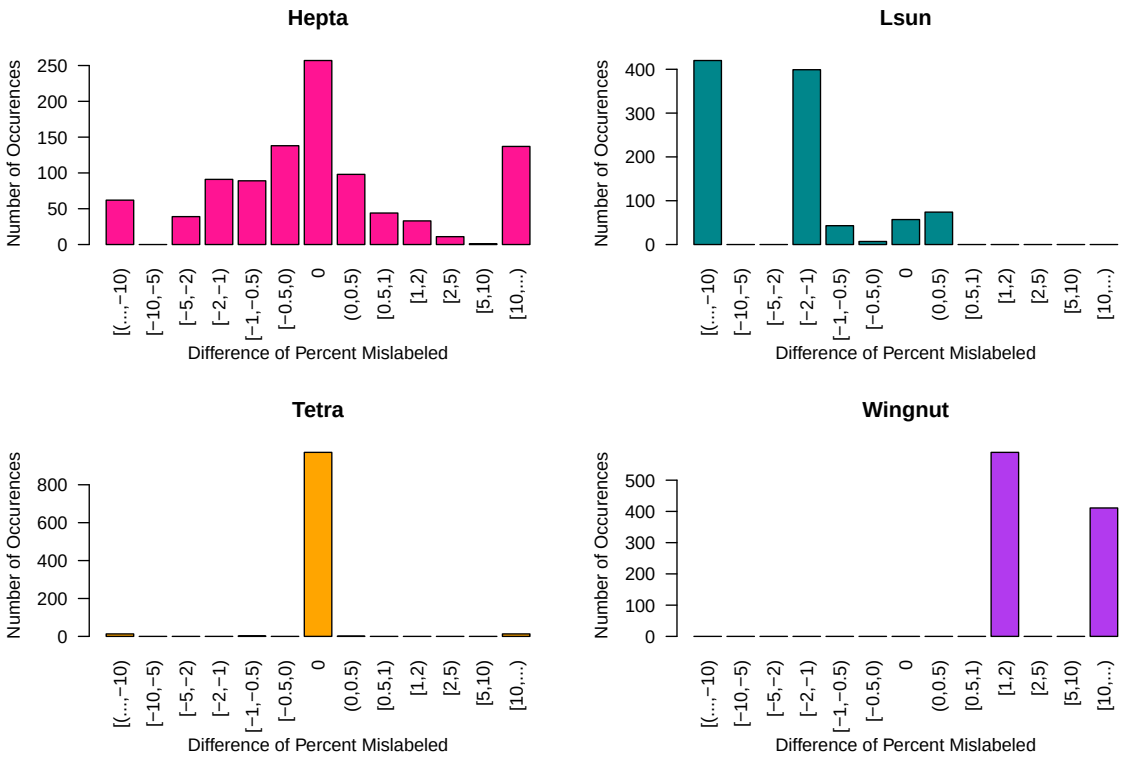

Figure 5.4: Difference in percent of data points mislabeled for L_1-norm compared to the L_2-norm $\big($(% mislabeled L_1) - (% mislabeled L_2), 40 rounds$\big)$.

We see that indeed, it is impossible to say which metric is better in terms of accuracy – for the Hepta dataset, the performance is very balanced, for the Lsun dataset, the L_1-norm performs much better, for the Tetra dataset, they nearly always perform exactly the same, and for the Wingnut dataset, the L_2-norm is consistently better. Thus, we will switch to the L_1-norm in our implementation, though we include diagrams for the L_2-norm as well to show that our accuracy results are not specific to the L_1-norm.

5.4.3 Implementing the K-Means-Algorithm via Fractional Encoding

Recall the Fractional Encoding from Section 4.1, where we encode the numerator and denominator separately and perform operations using the computation rules for fractions. This allows us to perform divisions on encrypted data, which is not normally possible in FHE computations. Thus, we can perform the K-Means-Algorithm as it is (except that we use the L_1- instead of the L_2-norm) on encrypted data, which we refer to as the *exact* version of the algorithm in contrast to the modified, more efficient versions that we will present later in Section 5.4.4. As a baseline, we will now examine the performance of this exact version. A runtime comparison between all versions of the K-Means-Algorithm will be presented in Section 5.4.6.

5.4.3.1 *Accuracy*

To see how the exact algorithm performs in terms of accuracy with respect to the number of iterations, we use the four datasets from Section 5.4.1.2. We ran the exact algorithm 1000 times (again with randomly chosen starting centroids in each run) for number of iterations $T = 5, 10, ..., 45, 50$, and for sake of completeness we include both distance metrics. These results were obtained by running the algorithms in unencrypted form. We first examine the effect of T on the exact version of the algorithm by looking at the average (over the 1000 runs) misclassification rate for both metrics. The result can be seen in Figure 5.5.

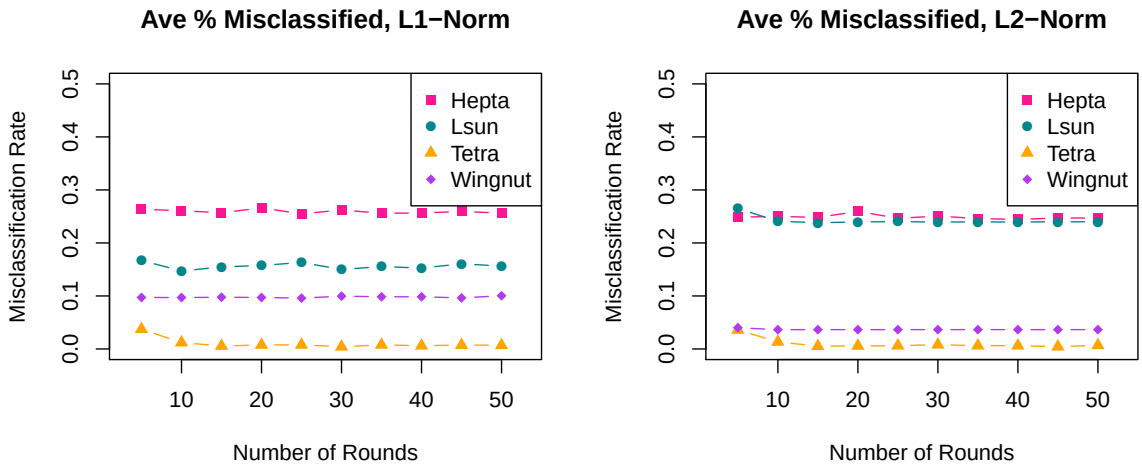

Figure 5.5: Misclassification rate with increasing rounds for the exact algorithm.

We can see that the rate levels off after about 15 rounds in all cases, so there is no reason to iterate further.

5.4.3.2 *Efficiency*

As already discussed, we need to shorten the bitlength to avoid a doubling of the length with each operation, which is not trivial to do. The initial data was encoded with the numerator in $[2^{11}, 2^{12})$ and denominator in roughly the same range, as the data is reasonably small. We also allotted 35 bits for nominator and denominator each to allow a growth in required bitlength, and set the shortening parameter to 12, but shortened by 11 every once in a while (we derived this approach experimentally, see the discussion of the shortcoming of this approach in Section 4.1.3). The fractional exact version of the K-Means-Algorithm was so slow that we could run it only on the first 10 data entries of the dataset and then extrapolated the runtimes in Section 5.4.6 (Table 5.1 on page 142) – running the K-Means-Algorithm on the Lsun dataset with Fractional Encoding would take almost 1.5 years on our computer. Because of this questionable performance and the issue with the shortening procedure, we instead chose to modify the K-Means-Algorithm to make it more FHE-friendly (i.e., remove the division), which we will present in the following.

5.4.4 The Stabilized K-Means-Algorithm

We now introduce a modification of the K-Means algorithm that avoids the division in the `MoveCentroid`-step. Concretely, recall that conventional encodings in FHE do not allow the computation of c_1/c_2 where c_1 and c_2 are ciphertexts, but it is possible to compute c_1/a where a is some unencrypted number. Our algorithm uses this fact to exchange the ciphertext division in Line 25 of Algorithm 16 for a constant division, resulting in a variant that can be computed with more established and efficient encodings than the Fractional Encoding – concretely, we use the Hybrid Encoding from Section 3.5. Note that this approach of approximating or replacing a function that is hard to compute on encrypted data is not unusual in the FHE context – for example, [GDL+16] does this for several different functions in building a neural network on encrypted data.

5.4.4.1 *Algorithm Details*

Recall that in the original K-Means-Algorithm, the `MoveCentroid`-step consists of computing each centroid as the average of all data entries that have been assigned to it. More specifically, suppose that we have a $(m \times K)$-dimensional cluster assignment matrix A, where

$$A_{ik} = \begin{cases} 1, & \text{Data entry } x_i \text{ is assigned to centroid } \mu_k \\ 0 & \text{else.} \end{cases}$$

Then computing the new centroid value μ_k can be written as multiplying the data entries x_i with the corresponding entry A_{ik} and summing up the results before dividing by the sum over the respective column k of A:

$$\mu_k = \frac{\sum_{i=1}^{m} x_i \cdot A_{ik}}{\sum_{i=1}^{m} A_{ik}}.$$

Our modification now replaces this procedure with the following idea: In order to compute the new centroid μ_k, we add the corresponding data entry x_i to the running sum if $A_{ik} = 1$, otherwise add the old centroid value $\bar{\mu_k}$ if $A_{ik} = 0$. This can be easily done with a multiplexer gate (applied to each bit of the two inputs) with the entry A_{ik} as the conditional boolean variable:

$$\mu_k = \frac{\sum_{i=1}^{m} \texttt{MUX}(A_{ik}, x_i, \bar{\mu_k})}{m}.$$

The sum now always consists of m terms, so we can divide by the unencrypted constant m. It is also now obvious why we call it the *stabilized* K-Means-Algorithm: We expect the centroids to move much more slowly, because the old centroid values stabilize the value in the computation (more so with fewer data entries that are assigned to a centroid). However, this new algorithm still maintains the spirit of the original K-Means-Algorithm: It is an iterative centroid-based algorithm that moves the centroids depending on the values of the data entries that are assigned to it, and the assignment is based on the smallest distance between data point and centroid. The details of this new algorithm can be found in Algorithm 18, with the changes compared to the original K-Means-Algorithm shaded.

Algorithm 18: The Stabilized K-Means-Algorithm

Input: Encrypted data set $X = \{x_1, \ldots, x_m\}$ // $x_i \in \mathbb{R}^\ell$ for some ℓ
Input: Number of clusters K (not encrypted)
Input: Number of iterations T (not encrypted)
// Initialization
Randomly reorder X
Set centroids $\mu_k = x_k$ for $k = 1$ to K
// Keep track of centroid assignments
Generate $(m \times K)$-dimensional boolean matrix A set to 0
for $j = 1$ *to* T **do**
  // Cluster Assignment
  for $i = 1$ *to* m **do**
    $\Delta = \infty$
    for $k = 1$ *to* K **do**
      // Compute distances to all centroids
      $\Delta_k := ||x_i - \mu_k||_1$
    end
    // The i^{th} row of A has all 0's except at the column corresponding to the centroid with the minimum distance
    $A[i, \cdot] \leftarrow \texttt{FindMin}(\Delta_1, \ldots, \Delta_K)$
  end
  // Move Centroids
  for $k = 1$ *to* K **do**
    // Keep old centroid value
    $\bar{\mu_k} = \mu_k$
    $\mu_k = 0$
    for $i = 1$ *to* m **do**
      // If $A_{ik} = 1$, add x_i to μ_k, otherwise add $\bar{\mu_k}$ to μ_k
      $\mu_k = \mu_k + \texttt{MUX}(A_{ik}, x_i, \bar{\mu_k})$
    end
    // Divide by number of terms m
    $\mu_k = \mu_k / m$
  end
end
Output: $\{\mu_1, \ldots, \mu_K\}$ (encrypted)

5.4.4.2 *Computing the Minimum*

As the reader may have noticed in Line 10, we have replaced the comparison step in finding the nearest centroid for a data entry with a new function $\texttt{FindMin}(\Delta_1, \ldots, \Delta_K)$ due the change in data structure of A (from an integer vector to a boolean matrix) and for readability. This new function outputs

$$A[i, \cdot] \leftarrow \texttt{FindMin}(\Delta_1, \ldots, \Delta_K)$$

such that the i^{th} row of A, $A[i, \cdot]$, has all 0's except at the column corresponding to the centroid with the minimum distance to x_i. The exact workings can be found in Algorithm 19.

Algorithm 19: $\texttt{FindMin}(\Delta_1, \ldots, \Delta_K)$

Input: Distances $\Delta_1, \ldots, \Delta_K$ of current data entry i to all centroids $\mu_1 \ldots, \mu_K$
Input: Row i of Cluster Assignment matrix A, denoted $A[i, \cdot]$
`// Set all entries 0 except the first`
1 Set $A[i, \cdot] = [1, 0, \ldots, 0]$
`// Set the minimum to` Δ_1
2 Set $\texttt{minval} = \Delta_1$
3 **for** $k = 2$ *to* K **do**
  `//` C `is a Boolean value,` $C = 1$ `iff` $\texttt{minval} \leq \Delta_k$
4 $C = \texttt{Compare}(\texttt{minval}, \Delta_k)$
5 **for** $r = 1$ *to* $k - 1$ **do**
    `// Set all previous values to 0 if new min is` Δ_k`, don't change if new min is old min`
6    $A[i, r] = A[i, r] \cdot C$
7 **end**
  `//` $\texttt{Set } A[i, k] \texttt{ to 1 if } \Delta_k \texttt{ is new min, 0 otherwise}$
8 $A[i, k] = C \oplus 1$
9 **if** $k \neq K$ **then**
    `// Update the minval variable unless we're done`
10    $\texttt{minval} = \texttt{MUX}(C, \texttt{minval}, \Delta_k)$
11 **end**
12 **end**
Output: $A[i, \cdot]$

The idea is to run the `Compare` circuit to obtain a Boolean value:

$$\texttt{Compare}(x, y) = \begin{cases} 1, & x < y, \\ 0, & x \geq y. \end{cases}$$

We start by comparing the first two distances Δ_1 and Δ_2 and setting the Boolean value as

$$C := \texttt{Compare}(\Delta_1, \Delta_2).$$

Then we can assign the first two entries of $A[i, \cdot]$ as

$$A[i, 1] = C \text{ and } A[i, 2] = C \oplus 1,$$

so there is a 1 in the position of the smaller of the two entries, and a 0 at the other one. We keep track of the current minimum through

$$\texttt{minval} := \texttt{MUX}(C, \Delta_1, \Delta_2).$$

We then compare `minval` to Δ_3 etc. until we have reached Δ_K. Note that we need to modify all entries $A[i,k]$ with k smaller than the current index by multiplying them with the current Boolean value C: If $C = 1$, the minimum does not change through the comparison, and all previous values are preserved. If the minimum does change, i.e., the current index has the smallest distance, then $C = 0$ and thus all previous values are set to 0.
We give a more formal proof of correctness:

Theorem 5.1. *The output* $A[i,\cdot] \leftarrow \textit{FindMin}(\Delta_1, \ldots, \Delta_K)$ *is correct, i.e., it has a* 1 *at the position of the minimum value, and* 0 *at all other positions.*

Proof: We prove this by induction: More specifically, we show that after iteration k ($k = 2, \ldots, K$) of the for-loop spanning lines 3 to 12 in Algorithm 19, $A[i,\cdot]$ is correct with respect to the first k positions – i.e., it has a 1 at the position of the minimum of these first k positions, and a 0 at all others. This means that after all $K-1$ iterations of the loop, the final result is correct, as we have reached the length of the entire array. Also, we show that the minimum value keeps track of the correct value.

For the beginning of our induction, we consider the first iteration of the loop, so $k = 2$. We started by initializing $A[i,\cdot] = [1, 0, \ldots, 0]$, and $\texttt{minval} = \Delta_1$. Now we compute $C = \texttt{Compare}(\texttt{minval}, \Delta_2) = \texttt{Compare}(\Delta_1, \Delta_2)$, so we have

$$C = \begin{cases} 1, & \Delta_1 < \Delta_2 \\ 0, & \Delta_1 \geq \Delta_2. \end{cases}$$

We then move on to the inner loop (lines 5 to 7), which just sets $A[i,1] = A[i,1] \cdot C$. After the inner loop, we set $A[i,2] = C \oplus 1$. To see correctness, we consider two cases:
Case 1: $C = 1 \Leftrightarrow \Delta_1 < \Delta_2$.
Then we have $A[i,1] = A[i,1] \cdot C = 1 \cdot 1 = 1$ and $A[i,2] = C \oplus 1 = 1 \oplus 1 = 0$, so the first two entries of $A[i,\cdot]$ are $[1,0]$.
Since $\Delta_1 < \Delta_2$, this is correct.
Updating the minimum value yields
$\texttt{minval} = \texttt{MUX}(C, \texttt{minval}, \Delta_2) = \texttt{MUX}(1, \Delta_1, \Delta_2) = \Delta_1$, which is also correct.
Case 2: $C = 0 \Leftrightarrow \Delta_1 \geq \Delta_2$.
Then we have $A[i,1] = A[i,1] \cdot 0 = 1 \cdot 0 = 0$ and $A[i,2] = C \oplus 1 = 0 \oplus 1 = 1$, so the first two entries of $A[i,\cdot]$ are $[0,1]$.
Since $\Delta_1 \geq \Delta_2$, this is correct.
Updating the minimum value yields
$\texttt{minval} = \texttt{MUX}(C, \texttt{minval}, \Delta_2) = \texttt{MUX}(0, \Delta_1, \Delta_2) = \Delta_2$, which is also correct.

For the induction step, suppose that we are in iteration k of the outer loop, so the first $k-1$ entries and `minval` are correct according to the induction assumption. Let k^* denote the position that is set to 1 in the first $k-1$ entries of $A[i,\cdot]$. Then we compute

$$C = \texttt{Compare}(\texttt{minval}, \Delta_k) = \begin{cases} 1, & \texttt{minval} < \Delta_k \\ 0, & \texttt{minval} \geq \Delta_k. \end{cases}$$

We again consider two cases:
Case 1: $C = 1 \Leftrightarrow \texttt{minval} < \Delta_k$.
This means that the minimum did not change, so the position of the minimum should remain at k^*.
In the inner loop, we compute $A[i,r] = A[i,r] \cdot C = A[i,r] \cdot 1 = A[i,r]$ for $r = 1, \ldots, k-1$.
Thus, the first $k-1$ entries do not change, so the 1 is still at position k^*.
After the inner loop, we compute $A[i,k] = C \oplus 1 = 1 \oplus 1 = 0$.
Thus, the first k positions of $A[i,\cdot]$ are all 0 except for position k^*, which is 1. This is correct.
Updating the minimum value yields
$\texttt{minval} = \texttt{MUX}(C, \texttt{minval}, \Delta_k) = \texttt{MUX}(1, \texttt{minval}, \Delta_k) = \texttt{minval}$, which is the old value and thus also correct.
Case 2: $C = 0 \Leftrightarrow \texttt{minval} \geq \Delta_k$.
This means that the minimum is now Δ_k, so the position of the minimum should change to k.
In the inner loop, we compute $A[i,r] = A[i,r] \cdot C = A[i,r] \cdot 0 = 0$ for $r = 1, \ldots, k-1$.
Thus, the first $k-1$ entries are now all 0.
After the inner loop, we compute $A[i,k] = C \oplus 1 = 0 \oplus 1 = 1$.
Thus, the first k positions of $A[i,\cdot]$ are all 0 except for position k, which is 1. This is correct.
Updating the minimum value yields
$\texttt{minval} = \texttt{MUX}(C, \texttt{minval}, \Delta_k) = \texttt{MUX}(0, \texttt{minval}, \Delta_k) = \Delta_k$, which is also correct.

Thus, when we reach the end of the outer loop, the result $A[i,\cdot]$ will be all 0 except for the position of the minimum, which will have a 1.

□

To make the this algorithm clearer, consider the following example:

Example 5.1: *Suppose we have*

$$x_i = 5, \mu_1 = 3, \mu_2 = 9, \mu_3 = 4 \text{ and } \mu_4 = 0.$$

Then the distance vector, i.e., the distance between x_i and the centroids, is

$$\Delta = (2, 4, 1, 5).$$

We start with

$$A_i = [1, 0, 0, 0]$$

and

$$\mathit{minval} = \Delta_1 = 2.$$

Round 1:
Compute

$$C = \mathit{Compare}(\mathit{minval}, \Delta_2) = \mathit{Compare}(2, 4) = 1.$$

Set

$$A_i[1] = A_i[1] \cdot C = 1 \cdot 1 = 1$$

and

$$A_i[2] = C \oplus 1 = 1 \oplus 1 = 0.$$

Set

$$\mathit{minval} = \mathit{MUX}(C, \mathit{minval}, \Delta_2) = \mathit{MUX}(1, 2, 4) = 2,$$

A_i *is now*

$$A_i = [1, 0, 0, 0].$$

Round 2:
Compute

$$C = \mathit{Compare}(\mathit{minval}, \Delta_3) = \mathit{Compare}(2, 1) = 0.$$

Set

$$A_i[1] = A_i[1] \cdot C = 1 \cdot 0 = 0,$$
$$A_i[2] = A_i[2] \cdot C = 0 \cdot 0 = 0,$$

and

$$A_i[3] = C \oplus 1 = 0 \oplus 1 = 1.$$

Set

$$\mathit{minval} = \mathit{MUX}(C, \mathit{minval}, \Delta_3) = \mathit{MUX}(0, 2, 1) = 1,$$

A_i *is now*

$$A_i = [0, 0, 1, 0].$$

Round 3:
Compute

$$C = \mathit{Compare}(\mathit{minval}, \Delta_4) = \mathit{Compare}(1, 5) = 1.$$

Set

$$A_i[1] = A_i[1] \cdot C = 0 \cdot 1 = 0,$$
$$A_i[2] = A_i[2] \cdot C = 0 \cdot 1 = 0,$$
$$A_i[3] = A_i[3] \cdot C = 1 \cdot 1 = 1,$$

and

$$A_i[4] = C \oplus 1 = 1 \oplus 1 = 0.$$

A_i *is now*

$$A_i = [0, 0, 1, 0].$$

This means that centroid 3 ($\mu_3 = 4$) *has the smallest distance to* $x_i = 5$, *which can be easily verified.* ———

Note that if the encryption scheme is one where multiplicative depth is important, it is easy to modify `FindMin` to be depth-optimal: Instead of comparing Δ_1 and Δ_2, then comparing the result to Δ_3, then comparing that result to Δ_4 etc., we could instead compare Δ_1 to Δ_2 and Δ_3 to Δ_4 and then compare those two results etc., reducing the multiplicative depth from linear in the number of clusters K to logarithmic.
Since depth is not important for our implementation choice TFHE (recall from Section 5.1.2 that the number of gates is the bottleneck), we implemented the function as described in Algorithm 19.
We will now investigate the performance of our Stabilized K-Means-Algorithm compared to the traditional K-Means-Algorithm.

5.4.4.3 *Accuracy*

The results in this subsection were obtained by running the algorithms in unencrypted form. As we are interested in relative accuracy as opposed to absolute accuracy, we merely care about the difference in the output of the modified and exact algorithms on the same input (i.e., datasets and starting centroids), not so much about the output itself. Recall that we obtained $T = 15$ as a good choice for number of rounds for the exact algorithm – however, as explained above, the cluster centroids converge more slowly in the stabilized version, so we will likely need more iterations here.
We now compare the performance of the stabilized version to the exact version. We perform this comparison by examining the average (over the 1000 iterations) difference in the misclassification rate. Thus, a value of 2 means that the stabilized version mislabeled 2% more instances than the exact version, and a difference of -1 means that the stabilized version miscassified 1% less data points than the exact version[4]. The results for both distance metrics can be seen in Figure 5.6.

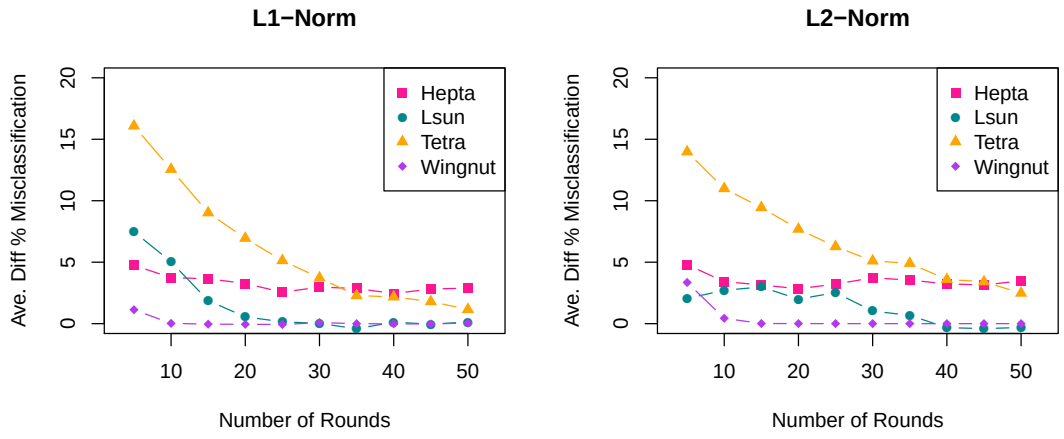

Figure 5.6: Average difference in misclassification rate between the stabilized and the exact algorithm ((average % mislabeled stabilized) - (average % mislabeled exact)).

[4]Note that Figure 5.6 does not really contain negative values – this is because the plotted values are averages, and on average the stabilized version usually does slightly worse than the original K-Means-Algorithm. However, in some individual instances, the stabilized version does perform better and thus yields a negative value, see e.g. Figure 5.7.

We see that while behavior varies slightly depending on the dataset, $T = 40$ iterations is a reasonable choice since the algorithms do not generally seem to converge further with more rounds. We will fix this parameter from here on, as it also exceeds the required amount of iterations for the exact version to converge.

As the reader may have noticed, while the values in Figure 5.6 do converge, they do not generally seem to reach a difference of 0, which would imply similar performance. However, this is not surprising – we did significantly modify the original algorithm, not with the intention of improving clustering accuracy, but rather to make it executable under an FHE scheme at all. This added functionality naturally comes as a tradeoff, and we will now examine the magnitude of the loss in accuracy in Figure 5.7 for the L_1-norm, and in Figure 5.8 for the L_2-norm.

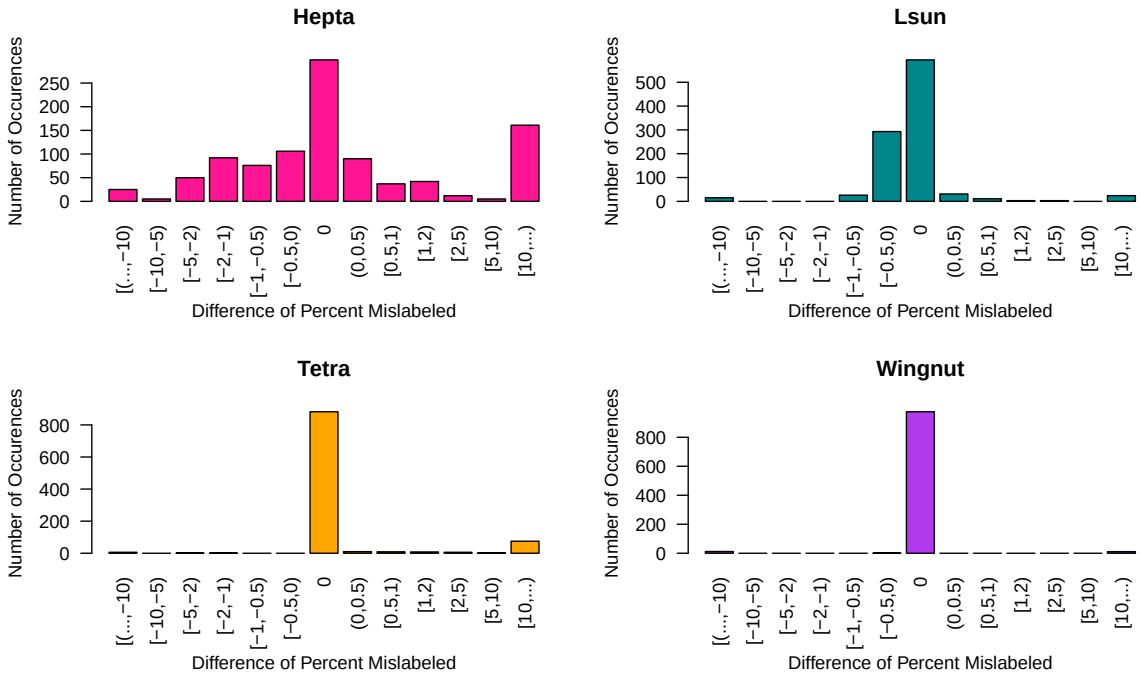

Figure 5.7: Distribution of the difference in misclassification rate for stabilized vs. exact K-Means-Algorithm $\big($(% mislabeled stabilized) - (% mislabeled exact)$\big)$, L_1-norm, 40 rounds.

We can see that in the vast majority of instances, the stabilized version performs exactly the same as the the original K-Means-Algorithm. We also see that concrete performance does depend on the dataset. In some cases, the modified version even outperforms the original one: Interestingly, for the Lsun dataset, the stabilized version is actually slightly better than the original algorithm in about 30% of the cases for the L_1-norm. However, we expect that most of the time, there will be a slight performance decrease. The fact that there are some outliers where performance is drastically worse can easily be solved by running the algorithm several times in parallel, and only keeping the best run. This can be done under homomorphic encryption much like computing the minimum in Section 5.4.4.2, but will not be implemented here.

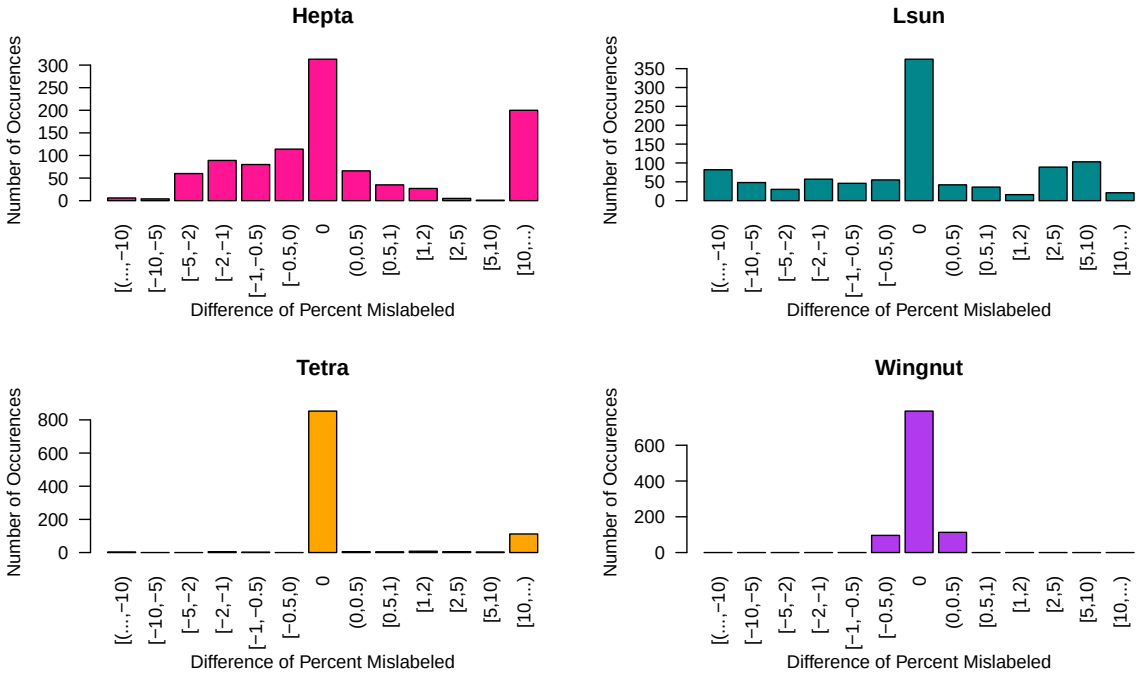

Figure 5.8: Distribution of the difference in misclassification rate for stabilized vs. exact K-Means-Algorithm $\big($(% mislabeled stabilized) - (% mislabeled exact)$\big)$, L_2-norm, 40 rounds.

5.4.4.4 *Efficiency*

While we will have a more detailed discussion of the runtime of all algorithms on encrypted data in Section 5.4.6, we would like to already present the performance gain at this point: Recall that we estimated that running the exact algorithm from Section 5.4.3 on encrypted data would take almost 1.5 years. In contrast, our Stabilized Algorithm can be run on encrypted data in 25.93 days, or less than a month. This is less than 5% of the runtime of the exact version. Note that this is single-thread computation time on our machine, which could be greatly improved by parallelization as detailed in Section 5.4.6 (though these improvements would apply to both algorithms, but we expect the ratio between the two algorithms to stay the same). In conclusion, we feel that by modifying the K-Means-Algorithm, we have traded a very small amount of accuracy for the ability to perform clustering on encrypted data in a more reasonable amount of time, which is a functionality that has not been achieved previously. The next section will deal with an idea to improve runtimes even more.

5.4.5 The Approximate Stabilized K-Means-Algorithm

Recall from Section 4.3.2 the idea of the approximate comparison: Since we have encoded our numbers bitwise, we can delete the S least significant bits, which corresponds to dividing the number by 2^S and truncating. Let $\tilde{X}$ denote this truncated version of a number X, and $\tilde{Y}$ that of a number Y. Then

$$\texttt{Compare}(\tilde{X}, \tilde{Y}) = \texttt{Compare}(X, Y) \text{ if } |X - Y| \geq 2^S,$$

and may or may not return the correct result if $|X - Y| < 2^S$. However, correspondingly, if the result is wrong, the centroid that is wrongly assigned to the data entry is no more than 2^S further from the data entry than the correct one. Since the `Compare` function is linear in the length of its inputs, speeding up this building block could make the entire computation more efficient.

5.4.5.1 *Algorithm Details*

In the stabilized K-Means-Algorithm, we merely replace the comparison that is utilized in the `FindMin` function (Line 4 in Algorithm 19) by this approximate comparison. We propose to pick an initial S and decrease it over the course of the algorithm, so that accuracy increases as we near the end.

5.4.5.2 *Accuracy*

For the following results, we scaled the data with the factor 2^{20} and truncated to obtain the input data. This means that for $S = 5$, a wrongly assigned centroid would be at most 2^5 further from the data entry than the correct centroid on the scaled data - or no more than 2^{-15} on the original data scale. We set $S = \min\{7, (T/5) - 1\}$ where T is the number of iterations, and reduce S by one every 5 rounds. We again examine the average (over 1000 iterations) difference in the misclassification rate to both the exact algorithm in Figure 5.9 and the stabilized algorithm in Figure 5.10.

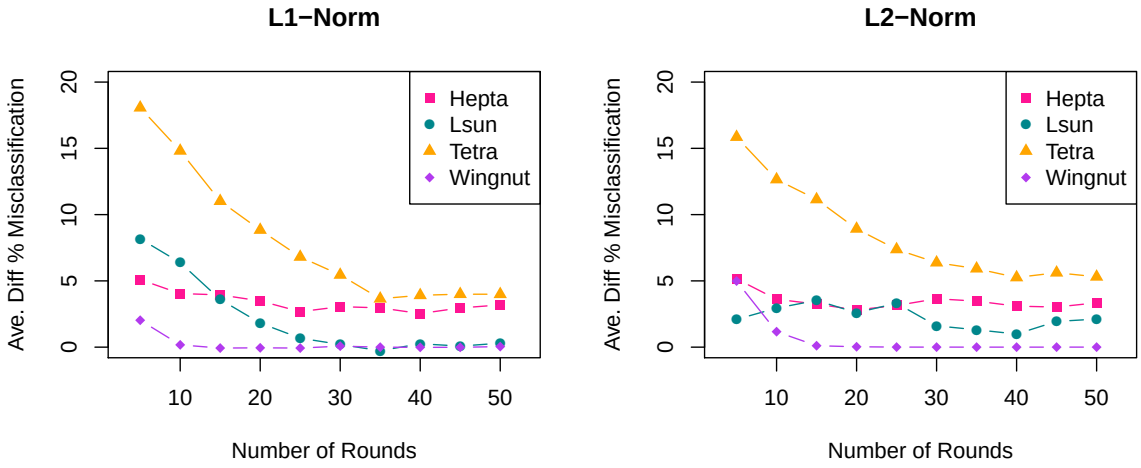

Figure 5.9: Average difference in misclassification rate between the approximate and the exact algorithm ((average % mislabeled approximate) - (average % mislabeled exact)).

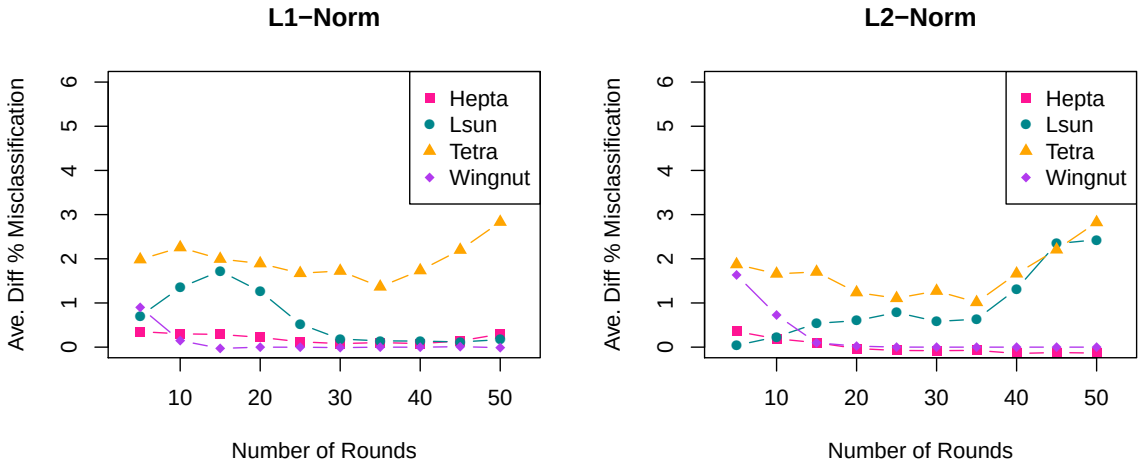

Figure 5.10: Average difference in misclassification rate for approximate vs. stabilized algorithm ((average % mislabeled approximate) - (average % mislabeled stabilized)).

We see that again, $T = 40$ rounds is a reasonable choice because the algorithms do not seem to converge further with more rounds – in fact, for some datasets, the performance of the approximate algorithm actually seems to decrease compared to the stabilized version (Figure 5.10) when the round number becomes very high. This may indicate that our shortening parameter $S = 7$ in the beginning was too high in these cases, which shows us that the optimal choice of S also depends on the underlying dataset and must be determined carefully.

We now again look at the distribution of difference in the misclassification rate. Figure 5.11 shows the distribution for the approximate versus the exact K-Means-Algorithm for the L_1-norm, and Figure 5.12 shows the same for the L_2-norm.

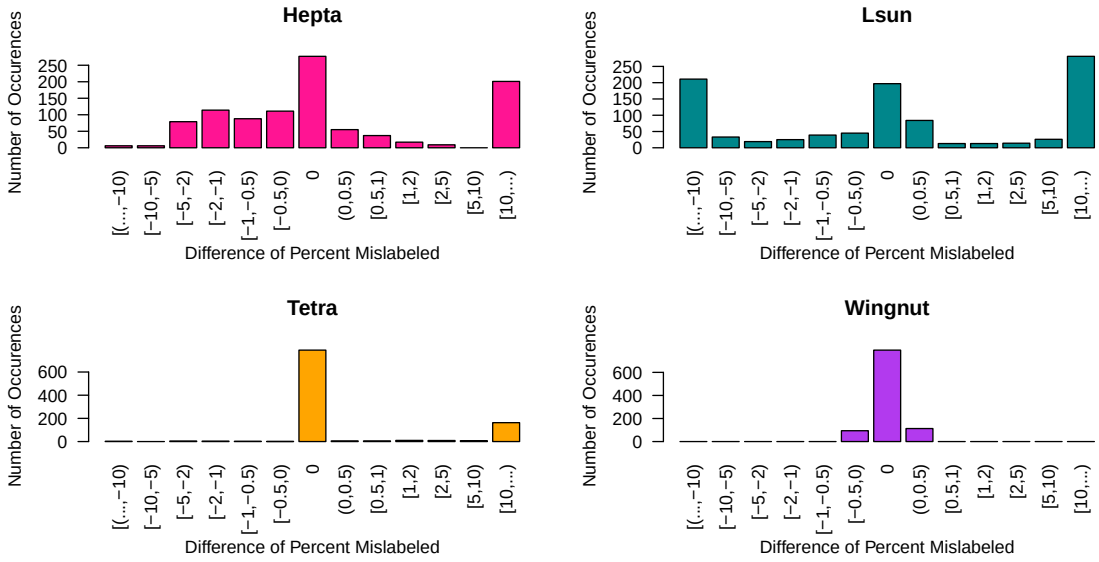

Figure 5.11: Distribution of the difference in misclassification rate for approximate vs. exact K-Means-Algorithm $\big($(% mislabeled approximate) - (% mislabeled exact)$\big)$, L_1-norm, 40 rounds.

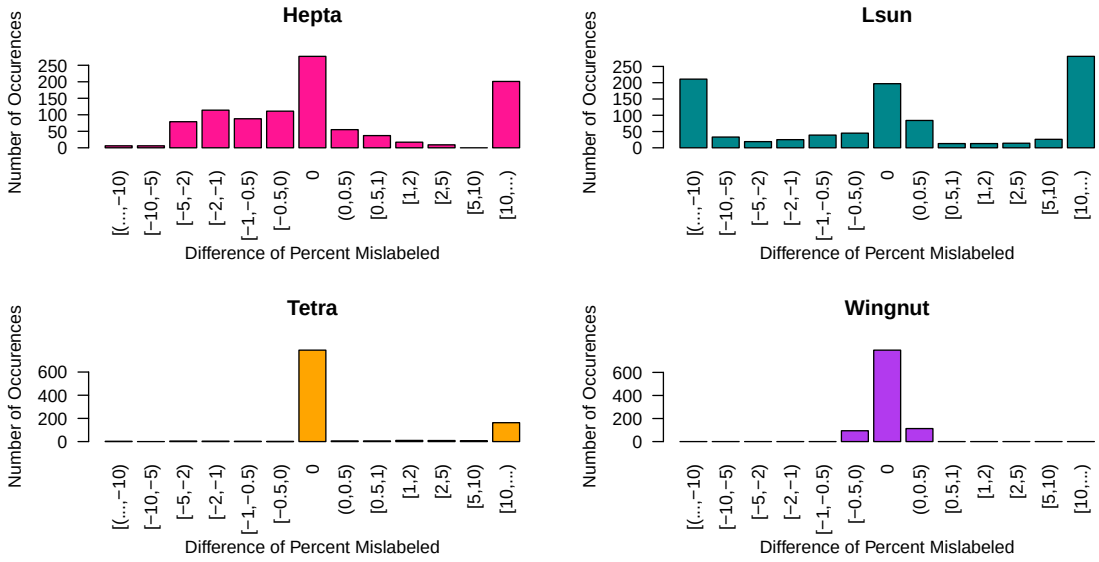

Figure 5.12: Distribution of the difference in misclassification rate for approximate vs. exact K-Means-Algorithm $\big($(% mislabeled approximate) - (% mislabeled exact)$\big)$, L_2-norm, 40 rounds.

We see that the approximate version is sometimes better and sometimes worse than the exact algorithm, but generally very close in performance, and that behavior depends on the dataset – in short, it looks a lot like the Figures 5.7 and 5.8, which compared the performance of the stabilized to the exact algorithm.
Thus, we lastly compare the distribution of difference in the misclassification rate between the approximate and stabilized versions of the algorithm in Figure 5.13 for the L_1-norm and Figure 5.14 for the L_2-norm .

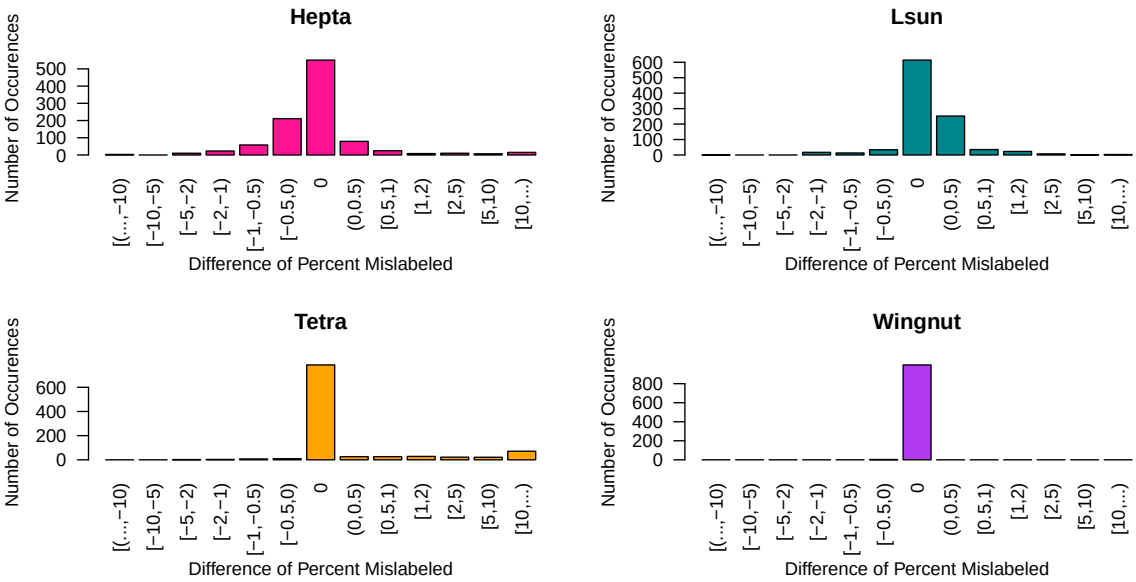

Figure 5.13: Distribution of the difference in misclassification rate for approximate vs. stabilized K-Means-Algorithm $\big($(% mislabeled approx.) - (% mislabeled stab.)$\big)$, L_1-norm, 40 rounds.

We see that for the vast majority of the cases, the performance of the approximate version is almost identical to the stabilized version or at most slightly worse. There is still the effect in the Lsun dataset that the approximate version outperforms the original K-Means-Algorithm in a significant amount of cases (though this effect mostly occurs for the L_1-norm), but it rarely does better than the stabilized version. This is not surprising, as it is in essence the stabilized version but with an opportunity for errors. However, because it is so close in performance to the stabilized version, we feel that this indeed seems to be a convenient way of trading in a small amount of accuracy for more efficiency.

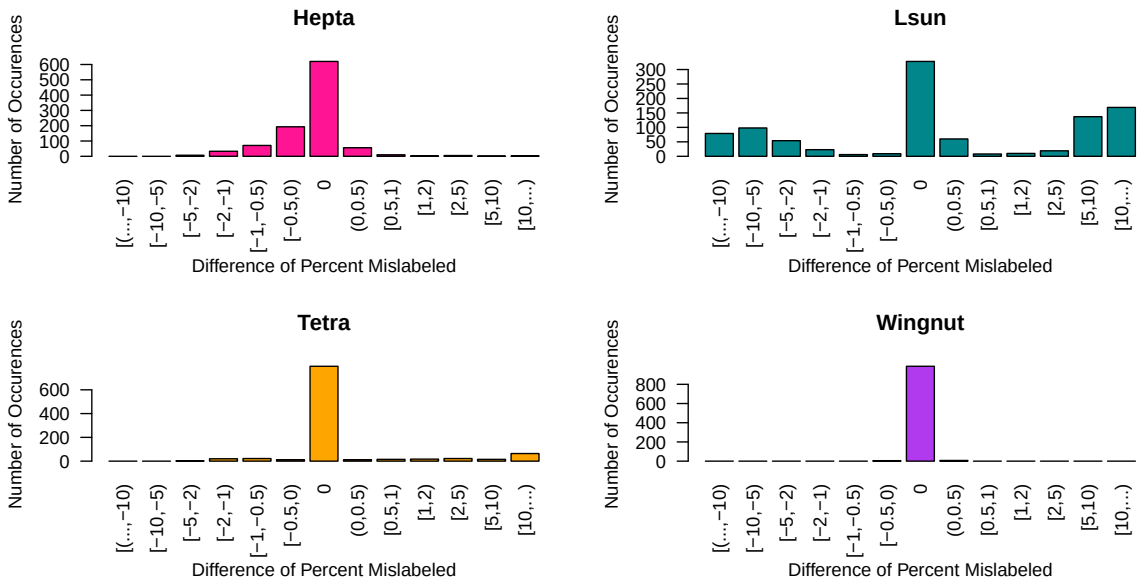

Figure 5.14: Distribution of the difference in misclassification rate for approximate vs. stabilized K-Means-Algorithm $\big($(% mislabeled approx.) - (% mislabeled stab.)$\big)$, L_2-norm, 40 rounds.

5.4.5.3 *Efficiency*

We now examine how much gain in terms of runtime we have from this modification. In our experiments, we set $S = 5$ so that accuracy is comparable to the stabilized version. Recall that it took about 1.5 years to run the exact algorithm with Fractional Encoding, and 25.93 days to run the stabilized version. The approximate version runs in 25.79 days, which means a difference of about 210.7 minutes. Of course, these runtimes are specific to the library [LIBf] we used – a different choice like [LIBd], where bit multiplication takes much longer than bit addition, may very well have a more significant improvement in its runtime. Unfortunately, for our implementation choice, the gain is swallowed by the magnitude of the total computation time, as the main bottleneck of the stabilized K-Means-Algorithm is actually the computation of the L_1-norm rather than the `FindMin`-procedure. Thus, for this specific application and implementation library, the approximate version may not be the best choice - however, for an algorithm that has a high number of comparisons relative to other operations, there can still be huge performance gains in terms of runtime: Recall from Section 4.3.2 that running just the comparison and approximate comparison functions with the same parameters as in our implementation of the K-Means-Algorithm (35 bits, 5 bits deleted for approximate comparison) yielded a drop in average (over 1000 runs each) runtime from 3.24 seconds for the regular comparison to 1.51 seconds for the approximate comparison. We see that this does make a big difference and may be of interest for computations involving many comparisons, which is why we choose to present the modification even though the effect was outweighed by other bottlenecks in our K-Means-Algorithm computation.

In conclusion, the approximate comparison provides the user with an easy method of trading in accuracy for faster computation, and most importantly, this loss of accuracy

can be decreased as computations near the end. However, for the specific application of the K-Means-Algorithm, these gains were unfortunately swallowed by the rest of the computation.

5.4.6 Performance

We now present the runtimes for the stabilized and approximate versions of the K-Means-Algorithm, along with the times for the exact version with the fractional encoding. Recall that the dataset we used was the Lsun dataset from [Ult05], which consists of 400 rational data entries of 2 dimensions, and has $K = 3$ clusters. We used the TFHE library, which currently supports only single-core computations, and we encoded the numbers by scaling them to integers with the factor 2^{20} and using Hybrid Encoding of length 35 bits. The timings we measured were for one round, and the approximate version used a deletion parameter of $S = 5$. For the Fractional Encoding, the initial data was encoded with the numerator in $[2^{11}, 2^{12})$ and denominator in roughly the same range. We also allotted 35 bits total for nominator and denominator each to allow a growth in required bitlength, and set the shortening parameter to 12, but shortened by 11 every once in a while. The fractional exact version was so slow that we ran it only on the first 10 data entries of the dataset – we will extrapolate the runtimes in Section 5.4.6.1.

5.4.6.1 *Runtimes for the Entire Algorithm on a Single Core*

In this subsection, we present the runtimes for the entire K-Means-Algorithm on encrypted data on our specific machine with single-thread computation. There is some extrapolation involved, as the measured runtimes were for one round (so we multiplied by the round number, which differs between the exact version and the other two, see Sections 5.4.3.1, 5.4.4.3 and 5.4.5.2), and in the fractional (exact) case, only for 10 data entries, so we multiplied that time by 40. Note that these times are with no parallelization, so there is much room for improvement as discussed in Section 5.4.6.2. The times can be found in Table 5.1.

	Exact (Fractional)	Stabilized	Approximate
Runtime per Round	873.46 hours $\approx$ 36.39 days	15.56 hours	15.47 hours
Rounds required	15	40	40
Total Runtime	545.91 days $\approx$ 17.95 months	25.93 days $\approx$ 0.85 months	25.79 days $\approx$ 0.85 months

Table 5.1: Single-thread runtimes (extrapolated) on encryped data on our machine.

We see that even though the stabilized version needs more rounds than the exact version, the latter is still significantly slower due to the Fractional Encoding. The approximate version (always with $S = 5$ deleted bits in the comparison) would save about 210.7 minutes (3.5 hours) here compared to the stabilized algorithm – unfortunately, this is only a small gain compared to the total runtime, as the bottlenecks in the encrypted K-Means-Algorithm computation lie elsewhere.

5.4.6.2 *Parallelism*

At this point, we would like to address the subject of parallelism. At the moment (last accessed May 29^{th} 2018), the TFHE library only supplies single-thread computations - i.e., there is no parallelism. However, version 1.5 is expected soon, and this will allegedly support multithreading. We first explain why this would make an enormous difference for the runtime, and then quantify the involved timings.
Looking at all our versions of the K-Means-Algorithm, it is easy to see that they are highly parallelizable: The `Cluster Assignment` step trivially so over the data entries (without any time needed for recombination of the individual results), and the `Move Centroids` similarly over the cluster centroids (the latter could also be parallelized over the data entries with a little recombination effort, which should still be negligible compared to the total running time). Since both steps are linear in the number K of centroids, the number m of data entries, and the number T of round iterations, we thus present the following runtimes as *per centroid, per data entry, per round, per core*. This allows a more flexible estimate for when multithreading is supported, as the ability to actually use our 4 allotted cores would lead to only about 1/4 of the total runtimes presented in Section 5.4.6.1.

5.4.6.3 *Round Runtimes*

We now present the runtime results for each of the three variants on encrypted data per centroid, per data entry, per round, per core in Table 5.2. We do not include runtimes for encoding/encryption and decryption/decoding, as these would be performed on the user side, whereas the computation would be outsourced (encoding/encryption is ca. 1.5 seconds, and decoding/decryption is around 5 ms).

	Exact (Fractional)	Stabilized	Approximate
`Cluster Assignment`	1650.91 s $\approx$ 27.5 min	35.59 s	35.39 s
`Move Centroids`	969.47 s $\approx$ 16.2 min	11.09 s	11.03 s
Total	2620.38 s $\approx$ 43.7 min	46.68 s	46.42 s

Table 5.2: Runtimes per centroid, per data entry, per round, per core on encrypted data.

We see that the Fractional Encoding is extremely slow, which motivated the Stabilized Algorithm in the first place. The approximate algorithm is faster than the stabilized version, but the difference is small, as discussed previously.

5.5 CONCLUSION

In conclusion, we have shown for different types of algorithms (training phase, classification phase and unsupervised learning) from the field of machine learning how to implement them on encrypted data. We have applied the results from the previous chapters, like Hybrid Encoding, length bounding, and the approximate comparison, to showcase their effect in improving the performance of these Machine Learning algorithms on encrypted data. For the K-Means-Algorithm, we changed the underlying algorithm in order to evaluate it on encrypted data in an efficient manner, resulting in a more FHE-friendly variant with very similar accuracy to the original and only 5% of its runtime. We thus saw the different approaches that can be taken when combining these two interesting fields of research, and many of the building blocks we used may be of independent interest for other applications of Fully Homomorphic Encryption.

CONCLUSION AND FUTURE RESEARCH

We have seen in this work that the choice of encoding makes a big difference in terms of efficiency when performing FHE computations. We saw in Chapter 2 that both in terms of additions and multiplications (and thus also total operations), it is best to choose the field $GF(2)$ as the encoding base in p-adic encoding. If multiplicative depth is the metric of choice, the optimal encoding base depends on the length of the involved numbers and the specific function to be applied, so there is no generic optimum. Extending the notion of p-adic encoding to the base field $GF(p^k)$ for $k \geq 1$, we saw that there is never a situation where choosing $k > 1$ is beneficial, as the performance for all metrics is worse than both encoding bases $GF(p)$ and $GF(p')$ with p' roughly the same size as p^k.

In Chapter 3, we saw how to incorporate negative numbers, and examined the effort incurred by the two most common encodings, Two's Complement and Sign-Magnitude. We learned that Two's Complement performs better when addding two numbers, but Sign-Magnitude is superior for multiplying two numbers. To bridge this gap, we showed how to switch from one encoding to the other and used this to construct a new encoding, called Hybrid Encoding, which essentially replaces the costly Two's Complement multiplication by switching to Sign-Magnitude, multiplying there, and switching back. This new multiplication procedure is only slightly more costly than Sign-Magnitude multiplication, and vastly outperforms Two's Complement multiplication even for very small bitlengths.

To include rational numbers in our computations, Chapter 4 first presented a Fractional Encoding, where numerator and denominator are separately encoded as integers. However, because this encoding has some downsides, we then showed how to scale rationals to integers so that the scaling factor and the precision stay constant throughout arbitrary computations, enabling us to then apply all contributions from previous chapters to these numbers. We also introduced other optimizations like length management to prevent quadratic growth of the bitlength with each multiplication, and ways to perform a comparison much faster if we accept a small chance of error.

Chapter 5 applied these contributions to algorithms from the field of Machine Learning, showcasing the performance gain through our improvements while simultaneously translating the algorithms (the Linear Means Classifier and the Perceptron) into the FHE building blocks derived in the previous chapter. Additionally, we took the K-Means-Algorithm, which cannot be dissected into FHE-friendly functions because there is a division involved, and instead changed the algorithm. We saw that the accuracy of the new algorithm is comparable to the original K-Means-Algorithm, and in terms of runtime it far outperforms Fractional Encoding. We have thus incorporated different types of Machine Learning algorithms into the FHE context – an important endeavor in a time where Machine Learning is becoming more and more popular, yet awareness for data privacy is simultaneously rising.

For future work, we see two main directions: Firstly, it would be interesting to further expand the analysis of different encodings (not restricted to p-adic encoding) and also incorporate more advanced algorithms, especially for multiplication. In unencrypted computations, there are many algorithms that perform much faster than the schoolbook multiplication we use, but many rely on conditional `IF-THEN` statements. Since we cannot see the conditional if it is encrypted, we always need to compute all conditional branches

and combine them via a multiplexer, which usually makes them prohibitively complicated. However, this need not hold for all of these algorithms – for example, the recursive Karatsuba Multiplication Algorithm seems promising in this context. The second field of interest would be to incorporate new classes of Machine Learning algorithms into the FHE context, possibly changing the algorithms as necessary while preserving performance – possible choices would be other clustering algorithms, outlier detection, image recognition, or online learning.

Bibliography

[ABC+15] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A. Reuter, and Martin Strand, *A guide to fully homomorphic encryption*, IACR Cryptology ePrint Archive (1192), 2015.

[AEH15] Louis J. M. Aslett, Pedro M. Esperança, and Chris C. Holmes, *Encrypted statistical machine learning: new privacy preserving methods*, CoRR abs/1508.06845, 2015.

[AHK01] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim, *On the surprising behavior of distance metrics in high dimensional spaces*, ICDT, 2001.

[AN16] Seiko Arita and Shota Nakasato, *Fully homomorphic encryption for point numbers*, IACR Cryptology ePrint Archive (402), 2016.

[AP14] Jacob Alperin-Sheriff and Chris Peikert, *Faster bootstrapping with polynomial error*, CRYPTO, 2014.

[AS11] Frederik Armknecht and Thorsten Strufe, *An efficient distributed privacy-preserving recommendation system*, Med-Hoc-Net, 2011.

[BBB+17] Charlotte Bonte, Carl Bootland, Joppe W. Bos, Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren, *Faster homomorphic function evaluation using non-integral base encoding*, IACR Cryptology ePrint Archive (333), 2017.

[BBL17] Daniel Benarroch, Zvika Brakerski, and Tancrède Lepoint, *FHE over the integers: Decomposed and batched in the post-quantum regime*, PKC, 2017.

[BCG+17] Christina Boura, Ilaria Chillotti, Nicolas Gama, Dimitar Jetchev, Stanislav Peceny, and Alexander Petric, *High-precision privacy-preserving real-valued function evaluation*, FC, 2017.

[BF11] Dan Boneh and David Mandell Freeman, *Homomorphic signatures for polynomial functions*, EUROCRYPT 2011, 2011.

[BGH13] Zvika Brakerski, Craig Gentry, and Shai Halevi, *Packed ciphertexts in lwe-based homomorphic encryption*, PKC, 2013.

[BGI+01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang, *On the (im)possibility of obfuscating programs*, CRYPTO, 2001.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan, *(leveled) fully homomorphic encryption without bootstrapping*, ITCS, 2012.

[BLLN13] Joppe W. Bos, Kristin E. Lauter, Jake Loftus, and Michael Naehrig, *Improved security for a ring-based fully homomorphic encryption scheme*, IMA, 2013.

[BLN14] Joppe W. Bos, Kristin E. Lauter, and Michael Naehrig, *Private predictive analysis on encrypted medical data*, Journal of Biomedical Informatics, vol. 50, 2014.

[BMMP17] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier, *Fast homomorphic evaluation of deep discretized neural networks*, IACR Cryptology ePrint Archive (1114), 2017.

[BO07] Paul Bunn and Rafail Ostrovsky, *Secure two-party k-means clustering*, CCS, 2007.

[BPHJ14] Christoph Bösch, Andreas Peter, Pieter H. Hartel, and Willem Jonker, *SOFIR: securely outsourced forensic image recognition*, ICASSP, 2014.

[BPP00] Joan Boyar, René Peralta, and Denis Pochuev, *On the multiplicative complexity of boolean functions over the basis ($\cap$, $\oplus$, 1)*, Theor. Comput. Sci., 2000.

[BPTG15] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser, *Machine learning classification over encrypted data*, NDSS, 2015.

[Bra12] Zvika Brakerski, *Fully homomorphic encryption without modulus switching from classical gapsvp*, CRYPTO, 2012.

[BSS+17] Anthony Barnett, Jay Santokhi, Michael Simpson, Nigel P Smart, Charlie Stainton-Bygrave, Srnivas Vivek, and Adrian Waller, *Image classification using non-linear support vector machines on encrypted data*, IACR Cryptology ePrint Archive (857), 2017.

[BSW11] Dan Boneh, Amit Sahai, and Brent Waters, *Functional encryption: Definitions and challenges*, TCC, 2011.

[BV11a] Zvika Brakerski and Vinod Vaikuntanathan, *Efficient fully homomorphic encryption from (standard) LWE*, FOCS, 2011.

[BV11b] Zvika Brakerski and Vinod Vaikuntanathan, *Fully homomorphic encryption from ring-lwe and security for key dependent messages*, CRYPTO, 2011.

[BV14] Zvika Brakerski and Vinod Vaikuntanathan, *Lattice-based FHE as secure as PKE*, ITCS, 2014.

[BV18] Charlotte Bonte and Frederik Vercauteren, *Privacy-preserving logistic regression training*, IACR Cryptology ePrint Archive (233), 2018.

[CCK+13] Jung Hee Cheon, Jean-Sébastien Coron, Jinsu Kim, Moon Sung Lee, Tancrède Lepoint, Mehdi Tibouchi, and Aaram Yun, *Batch fully homomorphic encryption over the integers*, EUROCRYPT, 2013.

[CdWM+17] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff, *Privacy-preserving classification on deep neural network*, IACR Cryptology ePrint Archive (035), 2017.

[CG15] Yao Chen and Guang Gong, *Integer arithmetic over ciphertext and homomorphic data aggregation*, CNS, 2015.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène, *Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds*, ASIACRYPT, 2016.

[CGGI17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène, *Improving TFHE: faster packed homomorphic operations and efficient circuit bootstrapping*, IACR Cryptology ePrint Archive (430), 2017.

[CGH+18] Jack L.H. Crawford, Craig Gentry, Shai Halevi, Daniel Platt, and Victor Shoup, *Doing real work with fhe: The case of logistic regression*, IACR Cryptology ePrint Archive (202), 2018.

[CK16] HeeWon Chung and Myungsun Kim, *Encoding rational numbers for fhe-based applications*, IACR Cryptology ePrint Archive (344), 2016.

[CKKS16] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song, *Homomorphic encryption for arithmetic of approximate numbers*, IACR Cryptology ePrint Archive (421), 2016.

[CKV10] Kai-Min Chung, Yael Tauman Kalai, and Salil P. Vadhan, *Improved delegation of computation using fully homomorphic encryption*, CRYPTO, 2010.

[CLP17] Hao Chen, Kim Laine, and Rachel Player, *Simple encrypted arithmetic library - SEAL v2.1*, IACR Cryptology ePrint Archive (224), 2017.

[CLPX17] Hao Chen, Kim Laine, Rachel Player, and Yuhou Xia, *High-precision arithmetic in homomorphic encryption*, IACR Cryptology ePrint Archive (809), 2017.

[CLT14] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi, *Scale-invariant fully homomorphic encryption over the integers*, PKC, 2014.

[CMNT11] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi, *Fully homomorphic encryption over the integers with shorter public keys*, CRYPTO, 2011.

[CNT12] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi, *Public key compression and modulus switching for fully homomorphic encryption over the integers*, EUROCRYPT, 2012.

[CSVW16] Anamaria Costache, Nigel P. Smart, S. Vivek, and A. Waller, *Fixed point arithmetic in SHE scheme*, IACR Cryptology ePrint Archive (250), 2016.

[DGBL+15] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing, *Manual for using homomorphic encryption for bioinformatics*, Tech. Report MSR-TR-2015-87, Microsoft Research, 2015.

[DM15] Léo Ducas and Daniele Micciancio, *FHEW: bootstrapping homomorphic encryption in less than a second*, EUROCRYPT, 2015.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias, *Multiparty computation from somewhat homomorphic encryption*, CRYPTO, 2012.

[Dwo06] Cynthia Dwork, *Differential privacy*, ICALP, 2006.

[EAH17] Pedro M. Esperança, Louis J. M. Aslett, and Chris C. Holmes, *Encrypted accelerated least squares regression*, AISTATS, 2017.

[FV12] Junfeng Fan and Frederik Vercauteren, *Somewhat practical fully homomorphic encryption*, IACR Cryptology ePrint Archive (144), 2012.

[Gam84] Taher El Gamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, CRYPTO, 1984.

[GC14] Matthias Geihs and Daniel Cabarcas, *Efficient integer encoding for homomorphic encryption via ring isomorphisms*, LATINCRYPT, 2014.

[GDL+16] 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*, ICML, 2016.

[Gen09] Craig Gentry, *A fully homomorphic encryption scheme*, Ph.D. thesis, Stanford University, 2009.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno, *Non-interactive verifiable computing: Outsourcing computation to untrusted workers*, CRYPTO, 2010.

[GH11] Craig Gentry and Shai Halevi, *Implementing Gentry's fully-homomorphic encryption scheme*, EUROCRYPT, 2011.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart, *Homomorphic evaluation of the AES circuit*, CRYPTO, 2012.

[GLN12] Thore Graepel, Kristin E. Lauter, and Michael Naehrig, *ML confidential: Machine learning on encrypted data*, ICISC, 2012.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters, *Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based*, CRYPTO, 2013.

[GVW15] Sergey Gorbunov, Vinod Vaikuntanathan, and Daniel Wichs, *Leveled fully homomorphic signatures from standard lattices*, STOC, 2015.

[GW13] Rosario Gennaro and Daniel Wichs, *Fully homomorphic message authenticators*, ASIACRYPT, 2013.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman, *NTRU: A ring-based public key cryptosystem*, ANTS, 1998.

[HS15] Shai Halevi and Victor Shoup, *Bootstrapping for helib*, EUROCRYPT, 2015.

[JA16] Angela Jäschke and Frederik Armknecht, *Accelerating homomorphic computations on rational numbers*, ACNS, 2016.

[JA17] Angela Jäschke and Frederik Armknecht, *(Finite) field work: Choosing the best encoding of numbers for fhe computation*, CANS, 2017.

[JA18] Angela Jäschke and Frederik Armknecht, *Unsupervised machine learning on encrypted data*, In submission, 2018.

[JKM05] Somesh Jha, Louis Kruger, and Patrick D. McDaniel, *Privacy preserving clustering*, ESORICS, 2005.

[JPH13] Arjan Jeckmans, Andreas Peter, and Pieter H. Hartel, *Efficient privacy-enhanced familiarity-based recommender system*, ESORICS, 2013.

[JPWU10] Geetha Jagannathan, Krishnan Pillaipakkamnatt, Rebecca N. Wright, and Daryl Umano, *Communication-efficient privacy-preserving clustering*, Trans. Data Privacy, 2010.

[JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan, *Gazelle: A low latency framework for secure neural network inference*, IACR Cryptology ePrint Archive (073), 2018.

[JW05] Geetha Jagannathan and Rebecca N. Wright, *Privacy-preserving distributed k-means clustering over arbitrarily partitioned data*, SIGKDD, 2005.

[KGV16] Alhassan Khedr, P. Glenn Gulak, and Vinod Vaikuntanathan, *SHIELD: scalable homomorphic implementation of encrypted data-classifiers*, IEEE Trans. Computers, 2016.

[KL15] Miran Kim and Kristin E. Lauter, *Private genome analysis through homomorphic encryption*, IACR Cryptology ePrint Archive (965), 2015.

[KSK+18] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon, *Logistic regression model training based on the approximate homomorphic encryption*, IACR Cryptology ePrint Archive (254), 2018.

[KSW+18] Miran Kim, Yongsoo Song, Shuang Wang, Yuhou Xia, and Xiaoqian Jiang, *Secure logistic regression based on homomorphic encryption*, IACR Cryptology ePrint Archive (074), 2018.

[KT16] Eunkyung Kim and Mehdi Tibouchi, *FHE over the integers and modular arithmetic circuits*, CANS, 2016.

[LIBa] LIBRARY: Coron's DGHV, *https://github.com/coron/fhe*.

[LIBb] LIBRARY: FHEW, *https://github.com/lducas/FHEW*.

[LIBc] LIBRARY: FV vs YASHE, *https://github.com/tlepoint/homomorphic-simon*.

[LIBd] LIBRARY: HELIB, *https://github.com/shaih/HElib*.

[LIBe] LIBRARY: SEAL, *https://www.microsoft.com/en-us/research/project/simple-encrypted-arithmetic-library/*.

[LIBf] LIBRARY: TFHE, *https://tfhe.github.io/tfhe*.

[LJY+15] Xiaoyan Liu, Zoe L. Jiang, Siu-Ming Yiu, Xuan Wang, Chuting Tan, Ye Li, Zechao Liu, Yabin Jin, and Jun-bin Fang, *Outsourcing two-party privacy preserving k-means clustering protocol in wireless sensor networks*, MSN, 2015.

[LKS16] Wenjie Lu, Shohei Kawasaki, and Jun Sakuma, *Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data*, IACR Cryptology ePrint Archive (1163), 2016.

[LLN14] Kristin E. Lauter, Adriana López-Alt, and Michael Naehrig, *Private computation on encrypted genomic data*, LATINCRYPT, 2014.

[LN14] Tancrède Lepoint and Michael Naehrig, *A comparison of the homomorphic encryption schemes FV and YASHE*, AFRICACRYPT, 2014.

[LTV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan, *On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption*, STOC, 2012.

[M+67] James MacQueen et al., *Some methods for classification and analysis of multivariate observations*, Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, 1967.

[MB12] Fatima Meskine and Safia Nait Bahloul, *Privacy preserving k-means clustering: a survey research*, Int. Arab J. Inf. Technol., 2012.

[NK15] Koji Nuida and Kaoru Kurosawa, *(batch) fully homomorphic encryption over integers for non-binary message spaces*, EUROCRYPT, 2015.

[NLV11] Michael Naehrig, Kristin E. Lauter, and Vinod Vaikuntanathan, *Can homomorphic encryption be practical?*, CCSW, 2011.

[PAH+17] Le Trieu Phong, Yoshinori Aono, Takuya Hayashi, Lihua Wang, and Shiho Moriai, *Privacy-preserving deep learning via additively homomorphic encryption*, IACR Cryptology ePrint Archive (715), 2017.

[Pai99] Pascal Paillier, *Public-key cryptosystems based on composite degree residuosity classes*, EUROCRYPT, 1999.

[Pim] *Pima dataset (last accessed 01.06.2018): https://www.kaggle.com/uciml/pima-indians-diabetes-database/data*.

[RAD78] Ronald L Rivest, Len Adleman, and Michael L Dertouzos, *On data banks and privacy homomorphisms*, Foundations of secure computation, 1978.

[RC14] Kurt Rohloff and David Bruce Cousins, *A scalable implementation of fully homomorphic encryption built on NTRU*, FC Workshops, 2014.

[Ros57] Frank Rosenblatt, *The perceptron, a perceiving and recognizing automaton*, Cornell Aeronautical Laboratory, 1957.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Commun. ACM, 1978.

[SED+88] Jack W Smith, JE Everhart, WC Dickson, WC Knowler, and RS Johannes, *Using the adap learning algorithm to forecast the onset of diabetes mellitus*, Proceedings of the Annual Symposium on Computer Application in Medical Care, 1988.

[SS10] Damien Stehlé and Ron Steinfeld, *Faster fully homomorphic encryption*, ASIACRYPT, 2010.

[SS11] Damien Stehlé and Ron Steinfeld, *Making NTRU as secure as worst-case problems over ideal lattices*, EUROCRYPT, 2011.

[SV10] Nigel P. Smart and Frederik Vercauteren, *Fully homomorphic encryption with relatively small key and ciphertext sizes*, PKC, 2010.

[SV14] Nigel Smart and Frederik Vercauteren, *Fully homomorphic SIMD operations*, DCC, 2014.

[Ult05] Alfred Ultsch, *Clustering with SOM: U*C*, Proc. Workshop on Self-Organizing Maps, 2005.

[VC03] Jaideep Vaidya and Chris Clifton, *Privacy-preserving* k*-means clustering over vertically partitioned data*, SIGKDD, 2003.

[vDGHV10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan, *Fully homomorphic encryption over the integers*, EUROCRYPT, 2010.

[WFNL16] David J. Wu, Tony Feng, Michael Naehrig, and Kristin E. Lauter, *Privately evaluating decision trees and random forests*, PoPETs, no. 4, 2016.

[XCWF16] Chen Xu, Jingwei Chen, Wenyuan Wu, and Yong Feng, *Homomorphically encrypted arithmetic operations over the integer ring*, ISPEC, 2016.

[XHY+17] Kai Xing, Chunqiang Hu, Jiguo Yu, Xiuzhen Cheng, and Fengjuan Zhang, *Mutual privacy preserving k -means clustering in social participatory sensing*, IEEE Trans. Industrial Informatics, 2017.

[Yao82] Andrew Chi-Chih Yao, *Protocols for secure computations (extended abstract)*, FOCS, 1982.

Appendix A

OVERVIEW OF ALGORITHMS

For reader convenience, we repeat the building block algorithms from Chapter 3 and their costs here.

A.1 UNSIGNED NUMBERS

A.1.1 Multiplexing

A.1.1.1 *Computation*

$$\texttt{MUX}(c, a, b) = c \cdot a + (1 + c) \cdot b = c \cdot (a + b) + b.$$

A.1.1.2 *Effort*

	Field Additions	**Field Multiplications**	**Depth**
Total	2	1	$\max\{\texttt{depth}(a), \texttt{depth}(b), \texttt{depth}(c)\} + 1$

A.1.2 Comparison of Unsigned Numbers

A.1.2.1 *Computation*

Input: Natural number $a = a_n \dots a_1 a_0$
Input: Natural number $b = b_n \dots b_1 b_0$
// Set result to 0
1 $res = 0$
2 **for** $i = 0$ *to* n **do**
  // Set temp to 0 if $a_i \neq b_i$ and to 1 if $a_i = b_i$
3 $temp = a_i + b_i + 1$
  // If $temp = 1$ (inputs are equal), don't change res. If $temp = 0$ (inputs are unequal), set $res = b_i$
4 $res = \texttt{MUX}(temp, res, b_i)$
5 **end**
// $res = 1 \Leftrightarrow a < b$
Output: res

A.1.2.2 *Effort*

	Field Additions	Field Multiplications	Depth
Total	$4n$	n	n

A.1.3 Addition of Unsigned Numbers

A.1.3.1 *Computation*

$$c_i = a_i + b_i + r_i,\ r_i = (a_{i-1} + b_{i-1}) \cdot (a_{i-1} + r_{i-1}) + a_{i-1}.$$

A.1.3.2 *Effort*

	Additions	Multiplications	Depth
Total	$5n - 4$	n	n

A.1.4 Subtraction of Unsigned Numbers

A.1.4.1 *Computation*

1. Flip all bits of $b = b_{n-1} \dots b_1 b_0$ to obtain $\bar{b} = \bar{b}_{n-1} \dots \bar{b}_1 \bar{b}_0$.
2. Compute $a - b = a + \bar{b} + 1$ using unsigned addition and discarding the $(n+1)^{\text{th}}$ result bit.

A.1.4.2 *Effort*

	Additions	Multiplications	Depth
Total	$11n - 14$	$2n - 2$	$n - 1$

A.1.5 Multiplication of Unsigned Numbers

A.1.5.1 *Computation*

1. Compute $a_i \cdot b_j$ for all $i = 0, \ldots, n-1, j = 0, \ldots, m-1$ and arrange into multiplication matrix.
2. Sum up the rows.

A.1.5.2 *Effort*

With `NatAdd` denoting the addition of unsigned numbers:

$$\sum_{i=1}^{\lfloor \log_2(m) \rfloor} \left\lceil \frac{m}{2^i} \right\rceil \cdot \texttt{Eff}(\texttt{NatAdd}(n + 2^{i-1} + i - 2)),$$

where `Eff` is the number of additions or multiplications, respectively. The maximum depth is $m + n - 1$.

A.2 TWO'S COMPLEMENT

A.2.1 Addition in Two's Complement

A.2.1.1 *Computation*

Like unsigned addition, but with sign extension.

A.2.1.2 *Effort*

	Additions	**Multiplications**	**Depth**
Total	$5n - 2$	n	n

A.2.2 Multiplication in Two's Complement

A.2.2.1 *Computation*

Assume that our numbers have lengths m and n, respectively.

1. Increase the bitlength of both numbers through sign extension (as described above) to length $m + n$.
2. Perform regular binary multiplication of the two resulting numbers. Note that to add the individual rows, we must use the addition function from above.
3. Keep only the rightmost $n + m$ bits.

A.2.2.2 *Effort*

	Additions	**Multiplications**	**Depth**
Total	$\frac{5(m^2+n^2)-19(m+n)}{2} + 5mn + 10$	$m \cdot n + \frac{(n+m-2)\cdot(n+m-1)}{2}$	$m + n - 1$

A.2.3 Negation in Two's Complement

A.2.3.1 *Computation*

1. Flip all bits (i.e., XOR them with 1).
2. Add 1 to the resulting number using unsigned addition and discarding the $(n+1)^{\text{th}}$ result bit.

A.2.3.2 *Effort*

	Additions	**Multiplications**	**Depth**
Total	$6n - 7$	$n - 1$	$n - 1$

A.2.4 Comparison in Two's Complement

A.2.4.1 *Computation*

Let `NatComp` denote the comparison of unsigned binary numbers.

Input: Signed number $a = a_n \dots a_1 a_0$
Input: Signed number $b = b_n \dots b_1 b_0$
`// Compare as if natural numbers, result is correct if signs are equal`
1 $c = \texttt{NatComp}(a, b)$
`// The sign bits are` a_n `and` b_n
2 $res = a_n \cdot (b_n + 1) + (a_n + b_n + 1) \cdot c$
Output: res

A.2.4.2 *Effort*

	Field Additions	**Field Multiplications**	**Depth**
Total	$4n + 4$	$n + 2$	$n + 1$

A.3 SIGN-MAGNITUDE

A.3.1 Addition in Sign-Magnitude

A.3.1.1 *Computation*

Input: Signed number $a = a_n \dots a_1 a_0$
Input: Signed number $b = b_n \dots b_1 b_0$
// Get numbers without sign bits, i.e., $|a|$ and $|b|$.
1 $\tilde{a} = a_{n-1} \dots a_1 a_0$
2 $\tilde{b} = b_{n-1} \dots b_1 b_0$
// res_1 is the unsigned addition of $|a|$ and $|b|$ with a's sign bit.
3 $res_1 = a_n || \texttt{NatAdd}(\tilde{a}, \tilde{b})$
// res_2 is the unsigned subtraction of $|b| - |a|$ with b's sign bit.
4 $res_2 = b_n || \texttt{NatSub}(\tilde{b}, \tilde{a})$
// res_3 is the unsigned subtraction of $|a| - |b|$ with a's sign bit.
5 $res_3 = a_n || \texttt{NatSub}(\tilde{a}, \tilde{b})$
// $c_1 = 1$ if $a_n \neq b_n$, and 0 if the signs are equal.
6 $c_1 = a_n + b_n$
// $c_2 = 1$ if $\tilde{a} < \tilde{b}$, and 0 otherwise.
7 $c_2 = \texttt{NatComp}(\tilde{a}, \tilde{b})$
// $temp = res_2$ if $c_2 = 1$, and $temp = res_3$ otherwise.
8 $temp = \texttt{MUX}(c_2, res_2, res_3)$
// $res = temp$ if $c_1 = 1$, and $res = res_1$ otherwise.
9 $res = \texttt{MUX}(c_1, temp, res_1)$
Output: res

A.3.1.2 *Effort*

	Field Additions	**Field Multiplications**	**Depth**
Total	$35n - 60$	$8n - 9$	$n + 1$

A.3.2 Multiplication in Sign-Magnitude

A.3.2.1 *Computation*

Delete the sign bits a_n and b_n, multiply the results as unsigned integers, and append the sign bit $a_n + b_n$.

A.3.2.2 *Effort*

To multiply two numbers with n and m bits:

	Field Additions	**Field Multiplications**	**Depth**
Total	$1 + \sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \lceil \frac{m-1}{2^i} \rceil \cdot \left(5 \cdot (n + 2^{i-1} + i) - 19\right)$	$\sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \lceil \frac{m-1}{2^i} \rceil \cdot \left(n + 2^{i-1} + i - 3\right)$	$m + n - 3$

A.3.3 Negation in Sign-Magnitude

A.3.3.1 *Computation*

Flip the MSB (i.e., set $a_n = a_n + 1$).

A.3.3.2 *Effort*

	Field Additions	**Field Multiplications**	**Depth**
Total	1	0	0

A.3.4 Comparison in Sign-Magnitude

A.3.4.1 *Computation*

Denote the comparison of unsigned binary numbers as `NatComp`.

Input: Signed number $a = a_n \dots a_1 a_0$
Input: Signed number $b = b_n \dots b_1 b_0$
`// Get numbers without sign bits, i.e.,` $|a|$ `and` $|b|$`.`
1 $\tilde{a} = a_{n-1} \dots a_1 a_0$
2 $\tilde{b} = b_{n-1} \dots b_1 b_0$
`// Compare absolute values as natural numbers`
3 $c = \texttt{NatComp}(\tilde{a}, \tilde{b})$
`// The sign bits are` a_n `and` b_n
4 $res = a_n + (a_n + b_n + 1) \cdot c$
Output: res

A.3.4.2 *Effort*

	Field Additions	**Field Multiplications**	**Depth**
Total	$4n - 1$	n	n

A.4 HYBRID ENCODING

A.4.1 Switching

A.4.1.1 *Computation*

Two's Complement to Sign-Magnitude:

Input: Signed number $a = a_n \dots a_1 a_0$ in Two's Complement encoding
// Get the negative of a in Two's Complement encoding.
1 $\tilde{a} = \texttt{TCNeg}(a)$
// Get the negative of $\tilde{a}$ in Sign-Magnitude encoding.
2 $\bar{a} = \texttt{SMNeg}(\tilde{a})$
// Assign $\bar{a}$ to the result if a is negative ($a_n = 1$), and assign a otherwise.
3 $res = \texttt{MUX}(a_n, \bar{a}, a)$
Output: res

Sign-Magnitude to Two's Complement:

Input: Signed number $a = a_n \dots a_1 a_0$ in Sign-Magnitude encoding
// Get the negative of a in Sign-Magnitude encoding.
1 $\tilde{a} = \texttt{SMNeg}(a)$
// Get the negative of $\tilde{a}$ in Two's Complement encoding.
2 $\bar{a} = \texttt{TCNeg}(\tilde{a})$
// Assign $\bar{a}$ to the result if a is negative ($a_n = 1$), and assign a otherwise.
3 $res = \texttt{MUX}(a_n, \bar{a}, a)$
Output: res

A.4.1.2 *Effort*

	Field Additions	**Field Multiplications**	**Depth**
Total	$8n - 6$	$2n - 1$	n

A.4.2 Multiplication in Hybrid Encoding

A.4.2.1 *Computation*

Input: Signed number $a = a_n \dots a_1 a_0$ in Two's Complement encoding
Input: Signed number $b = b_m \dots b_1 b_0$ in Two's Complement encoding
`// Switch` a `and` b `to Sign-Magnitude encoding.`
1 $\tilde{a} =$ `SwitchTCSM`(a)
2 $\tilde{b} =$ `SwitchTCSM`(b)
`// Multiply the values.`
3 $temp =$ `SMMult`$(\tilde{a}, \tilde{b})$
`// Switch` $temp$ `back to Two's Complement encoding.`
4 $res =$ `SwitchSMTC`$(temp)$
Output: res

A.4.2.2 *Effort*

	Field Additions	**Field Multiplications**	**Depth**
Total	$\sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \left(\lceil \frac{m-1}{2^i} \rceil \cdot \left(5 \cdot (n + 2^{i-1} + i) - 19\right) \right) + 16(m+n) - 17$	$\sum_{i=1}^{\lfloor \log_2(m-1) \rfloor} \left(\lceil \frac{m-1}{2^i} \rceil \cdot \left(n + 2^{i-1} + i - 3\right) \right) + 4(m+n) - 3$	$m + n$