(Finite) Field Work: Choosing the Best Encoding of Numbers for FHE Computation

Angela Jäschke
Frederik Armknecht
jaeschke@uni-mannheim.de
armknecht@uni-mannheim.de
University of Mannheim

ABSTRACT

Fully Homomorphic Encryption (FHE) schemes are a powerful tool that allows arbitrary computations on encrypted data. This makes them a promising tool for a variety of use cases that require outsourcing the computation on privacy-sensitive data to untrusted parties. FHE schemes today operate over finite fields, while many use cases call for natural or even real numbers, requiring appropriate encoding of the data into the plaintext space supported by the encryption scheme. However, the choice of encoding can tremendously impact the overall effort of the computation on the encrypted data. Surprisingly, although the question of selecting the encoding is quite natural and arises immediately in practice, a comprehensive analysis is still missing.

In this work, we formally and experimentally investigate this question for applications that operate over integers and rational numbers based on a selection of natural metrics: the number of finite field additions, the number of finite field multiplications, and the multiplicative depth. Our results are partly constructive and partly negative: We show that for the first two metrics, an optimal choice does exist and we state it explicitly. However, we show likewise that regarding multiplicative depth, the parameters need to be chosen specific to the use-case, as there is no global optimum. Still, we show exactly how one can choose the best encoding depending on the use-case.

1 INTRODUCTION

Fully Homomorphic Encryption (FHE) describes a class of encryption schemes which allow arbitrary computation on encrypted data. This makes them a promising tool for a variety of use cases that require outsourcing the computation on privacy-sensitive data to untrusted parties. Typical applications, e.g. see [2, 24], are privacy preserving medical analysis, recommender systems, machine learning algorithms, genome analysis, biometric authentication and financial analysis, to name a few. All these applications function over data structures involving rational or integer numbers, which must somehow be embedded into the plaintext space of the appropriate FHE scheme. We point out that if the entire function is known beforehand, the parameters can often be chosen in a way such that the entire computation remains in the (extremely large) plaintext space. This approach (called *Somewhat Homomorphic Encryption, SHE)* is often faster, but loses the main advantage of FHE: The flexibility to execute *any* function on the encrypted data. Moreover, the function may not be fully known in advance. Thus, to incorporate cases like proprietary algorithms or functions that are not known at encryption time, we focus on true FHE that allows arbitrary functions.

Unfortunately, current FHE schemes are not yet sufficiently practical due to the fact that their algorithms are quite involved and hence inherently more time consuming. However, continuous improvements are being made, with some current schemes achieving bootstrapping (see below) in less than a second. Thus, while there is still a lot a long way to go, it seems as though FHE is slowly approaching the realm of real-world feasibility. Publicly available implementations like [21] and [7] make the technology accessible beyond a theoretical level.

Another, often overlooked aspect that strongly impacts the overall performance is *how* an FHE scheme is applied. For instance, most FHE schemes today operate over finite fields $GF(p^k)$ for an arbitrary prime p and $k \geq 1$, while many use cases call for natural or even real numbers, requiring appropriate encoding of the data into the plaintext space supported by the encryption scheme. The standard way of doing this for a plaintext space $\mathbb{Z}_p$ or related is to represent the number in p-adic fashion[1] and encrypt each digit separately. Of course, when one then wants to operate on (e.g., add or multiply) these encrypted numbers, the operation must be transformed into an operation on the single encrypted digits. These operations quickly incur many consecutive multiplications of encrypted digits, yielding a further effort penalty.

The reason is that all currently known FHE schemes are noise-based, where the noise masks the plaintext and grows with each operation – linearly with each ciphertext addition, and quadratically with multiplication. When the noise surpasses a certain threshold, decryption fails and the plaintext cannot be retrieved. To solve this problem, there exists a so-called *bootstrapping* operation, which removes some of the noise before the ciphertext becomes unreadable. Unfortunately, it is very computationally intensive, so that reducing the number of bootstrappings has a huge effect on the overall effort.

Possibly surprising is the enormous impact which the chosen encoding of the data into the plaintext can have in this context. For example, [22] observes an increase in runtime from 0.004 to over 120 seconds just by switching from "normal" computation over the rational numbers (i.e., no encoding in our sense) to binary encoding, which emulates an FHE plaintext space of $\{0, 1\}$ (without actually using any encryption). In fact, the same paper observes that the choice of encoding can tremendously impact the overall effort, incurring a nearly 50% longer runtime for the worst examined encoding compared to the best one. Thus, a user or company

[1] E.g., if the plaintext space is $\{0, 1\}$, the binary encoding for natural numbers.

that aims to use FHE immediately faces the following question:

"What is the preferable **encoding of my plaintext data** *such that later on, the operations on the encrypted data* **incur an overhead as small as possible?***"*

Although this question is quite natural and arises immediately in practice, a comprehensive analysis of the best encoding is still missing. In this work, we investigate this question for applications that operate over integers and rational numbers. To this end, we formally and experimentally analyze the effort for FHE computation subject to different encoding choices. We base our analysis on three natural cost metrics to measure the effort of performing some computation with FHE: number of additions of encrypted numbers, number of multiplications of encrypted numbers, and multiplicative depth required by the function to be executed. [2]

Our results are partly constructive and partly negative: We show that for the first two metrics, an optimal choice does exist and we state it explicitly. However, we show likewise that regarding multiplicative depth, the parameters should be chosen specific to the use-case, as there is no global optimum. Nonetheless, our formulas allow exactly to do this. Our contributions are:

Formula for adding two numbers in p^k-adic encoding
We first derive a generic formula that expresses how to add two numbers in p^k-adic encoding[3]. Although it seems to be quite natural to investigate the structure of this formula, we merely found [25] in open literature, which only covers a small subset of our work (see Section 8). Hence, this result may be of independent interest.

Cost analysis for natural numbers.
Based on the derived generic formula, we analyze the costs for adding two encrypted natural numbers in p^k-adic encoding. The cost metrics we examine are field additions, field multiplications, and multiplicative depth. Our results are as follows:
1) For p-adic encoding with respect to the required number of field additions or field multiplications, the efforts for additions and multiplications of encrypted integers strictly increase with p, making $p = 2$ by far the best choice. As a preview of our results, the increase of the required number of field additions and multiplications for adding two numbers of similar size depending on the choice of p can be seen in Figures 1a and 1b.
2) For p-adic encoding with respect to the depth metric, it likewise holds that the required depth grows *asymptotically* with increasing p. However for low primes, the required number of digits dominates the depth, making $p = 2$ a non-optimal choice. Unfortunately, the optimal choice of p when using only depth as a metric heavily depends on further use-case depended parameters, so there is no generic optimum. One parameter is the size of the encrypted data (more specifically, the required number of digits). Also, the optimal choice of p for multiplying two numbers is a different one than the optimal p for addition, so the best choice not only depends on the size of the input data, but also on the nature of the function one wants to apply. This can be seen in Figure 1c, with special attention directed to the high values near the left vertical axis. Still, our formulas allow to compute the best encoding in case that the use-case depending parameters mentioned above are known.
3) For p^k-adic encoding with $k > 1$, we show that performance is always worse compared to p-adic encoding.

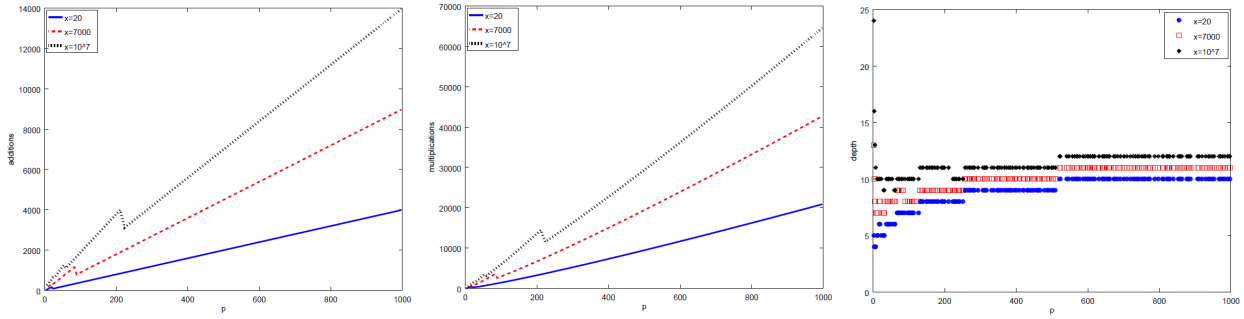
Cost analysis for rational numbers and integers
We then discuss how the analysis can be extended from natural numbers to rational numbers and integers. Through scaling (see Section 7.2), one can easily move from rational numbers to integers, but we quickly discuss how choosing the scaling factor relative to to the order of the finite field is particularly beneficial. Incorporating negative numbers proves to be the main challenge. We consider the two most common encodings for signed integers, p's Complement and Sign-Magnitude, and analyze their cost compared to natural numbers. We see that p's Complement encoding is a good choice for adding two numbers, and Sign-Magnitude encoding performs better for multiplication. We explain that by switching between these two encodings, we can get the lowest cost: Only slightly more than the same operation for natural numbers, in addition to the cost of switching. Thus, our results for natural numbers also extend both to integers and rational numbers, meaning that $p = 2$ is optimal with regard to field additions and multiplications, and depth does not have a generic optimum but the best choice can be computed depending on the use-case.

Outline. The paper is structured as follows: **Section 8** gives an overview of related work, and **Section 2** provides a discussion of the cost metrics considered in this work. In **Section 3**, we set about finding the formula for the carry values when adding two numbers in p-adic encoding. In **Section 4**, we analyze the effort for computing each digit of the result when adding two natural numbers, and in **Section 5** we use these results to compute the cost of adding or multiplying two natural numbers in p-adic encoding. **Section 6** presents the results of using $GF(p^k)$ for $k > 1$ as the encoding base. **Section 7** is concerned with extending the cost analysis to incorporate rationals and negative numbers. Lastly, **Section 9** gives our conclusion and briefly presents ideas for future work.

2 COST METRIC

In this section, we explain and motivate the cost metrics we base our analysis on. The majority of current FHE schemes support finite fields $GF(p^k)$ as plaintext space, making it necessary to embed the plaintext data into this structure. This means that a number A needs to be encoded by a set of digits $\tilde{a}_i \in GF(p^k)$.[4] Encrypting A essentially means encrypting each digit $\tilde{a}_i$, and computing on an encrypted number is accomplished by operating on the encrypted digits. That is, we need to emulate the original operation (e.g., adding two natural numbers) through finite field operations on the plaintext digits (i.e., a series of additions and multiplications on the individual digits) that result in a sequence of digits encoding the correct result of the addition. Consequently, we express the effort with regard to the underlying field operations although in reality, they would be performed on the encrypted digits in the ciphertext space, which depends on the specific encryption scheme being used.

[2] We argue in Section 2 why each of these cost metrics is relevant, but the preferred choice is essentially up to the reader.
[3] The term p^k-adic encoding denotes the natural extension of p-adic encoding to the field $GF(p^k)$ for $k \geq 1$ and is explained in Section 6.
[4] We will discuss typical encodings later in this paper.



(a) Additions (b) Multiplications (c) Depth

Figure 1: Number of field additions, multiplications and multiplicative depth for adding $x = 20/7000/10^7$ to a number of same size. The horizontal axis is the encoding base p, and the vertical axis the number of operations/depth, and the plots correspond to the three numbers.

We now present the three cost metrics that we use in the rest of the paper and briefly explain their respective relevance.

2.1 Multiplicative Depth

As explained in the introduction, all current FHE schemes are noise-based in that the plaintext is masked by noise, and each multiplication doubles the amount of noise. When a certain limit on the noise is passed, the ciphertexts cannot be decrypted correctly anymore. *Bootstrapping* can remove some of the noise before it exceeds this threshold, but this is a very costly operation. Thus, the goal is often to minimize the number of bootstrapping procedures that are necessary by minimizing the number of consecutive multiplications[5] (since each multiplication doubles the noise), also referred to as *multiplicative depth*. For this reason, multiplicative depth has been the standard cost metric and is naturally part of our analysis.

2.2 Number of Field Multiplications

The second cost metric we use to measure effort is the total number of (not necessarily consecutive) field multiplications. First off, multiplications of encrypted value are much more expensive than additions for all current schemes, so keeping track of this number seems like an obvious choice. In addition, the multiplicative depth one incurs in p-adic encoding quickly becomes so large that bootstrapping is unavoidable, so that minimizing the total number of multiplications can speed up performance significantly in this scenario. To see this, consider the extreme case of a parameter setting that requires bootstrapping after every multiplication: In this case, multiplicative depth is completely irrelevant and the number of multiplications determines the effort of the computation. This line of reasoning also extends to less extreme cases: For example, imagine a formula that requires the computation of a large number of terms to achieve the lowest multiplicative depth. If each of these terms requires a bootstrapping operation, but there is another way of computing the formula with a slightly higher depth and significantly fewer terms, the latter can have fewer total bootstrapping operations and thus be much faster. For this reason, we feel that minimizing depth mainly makes sense when it serves to avoid bootstrapping altogether – in cases where it is inevitable because even the optimal depth is too high, the number of multiplications may be the better cost metric.

[5]For example, suppose we need to do bootstrapping after 3 consecutive multiplications, and we want to compute $a \cdot b \cdot c \cdot d$ for some a, b, c, d. Then computing $(a \cdot b) \cdot (c \cdot d)$ only uses 2 consecutive multiplications and thus does not need bootstrapping, whereas computing $((a \cdot b) \cdot c) \cdot d$ has 3 consecutive multiplications and thus requires bootstrapping.

2.3 Number of Field Additions

For all schemes today, field additions cost almost nothing compared to field multiplications. However, there is no theoretical reason why this must be the case, so we include this metric because it might be valuable in the future for a different kind of scheme.

3 FORMULA FOR COMPUTING CARRY VALUES OVER $\mathbb{Z}_P$

In this section and the following Section 4, we lay the theoretical foundation for the effort analysis starting from Section 5. More concretely, we derive in this section the formulas for the digits of the sum of two numbers in p-adic encoding. As we will see, the carry digits are particularly important here so that we investigate them more closely in Section 4.

3.1 Overview

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

$$\begin{array}{cccccccc} & & a_n & a_{n-1} & \dots & a_2 & a_1 & a_0 \\ + & & b_n & b_{n-1} & \dots & b_2 & b_1 & b_0 \\ \hline = & c_{n+1} & c_n & c_{n-1} & \dots & c_2 & c_1 & c_0 \end{array} \quad (1)$$

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

$$c_i = c_i(a_n, b_n, a_{n-1}, b_{n-1}, ..., a_1, b_1, a_0, b_0) \quad (2)$$

for any i, where $c_i(\dots)$ refers to some polynomial (slightly abusing notation). Clearly, it holds that $c_i = a_i + b_i + r_i$, where $r_0 = 0$,

and for $i > 0$, r_i is the *carry* from position $i-1$. Our goal in this section is to express $r_i(a_{i-1}, b_{i-1}, r_{i-1})$ as a polynomial, which will constitute Theorem 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.

3.2 Formula

Recall that our goal is to find the formula from Equation 2. We have already established that $c_i = a_i + b_i + r_i$, where $r_0 = 0$, and for $i > 0$, r_i is the carry from position $i-1$. Thus, we now aim to find the polynomial that expresses the carry as a function of the inputs and carry from the previous position: $r_i(a_{i-1}, b_{i-1}, r_{i-1})$. To simplify the notation, we define the following expression:

Definition 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 can be seen in the proof of Theorem 1 in Appendix A. We now state the formula for the carry using these $l_i(x)$-functions:

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

$$\begin{aligned} r_i(a, b, r) &= \sum_{k=1}^{p-1} \left(l_k(b) \cdot \sum_{j=1}^{k} l_{p-j}(a) \right) + r_{i-1} \cdot (p-1) \cdot l_{p-1}(a+b) \\ &= f_1(a, b) + r_{i-1} \cdot f_2(a, b). \end{aligned}$$

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 $a_{i-1}, b_{i-1} \in \{0, \ldots, p-1\}, r_{i-1} \in \{0, 1\}$.

The proof is given in Appendix A.

4 THE EFFORT OF COMPUTING THE CARRY

4.1 Overview

In this section, we prove that the effort required to compute each digit c_i when adding two natural numbers encoded p-adically is

- Field additions: $5p - 4$
- Field multiplications: $2p \cdot \log_2(p) + 2p - 2 \cdot \log_2(p) - 4$
- Multiplicative depth: $\lceil \log_2(p) \rceil + i - 1$

To this end, we make use of Theorem 1 from Section 3. 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 1). Putting this together, we get as effort for computing c_i for $i > 1$ (where $\mathtt{D}(f_i)$ denotes the multiplicative depth of f_i, $\mathrm{Adds}(f_i)$ denotes the number of field additions incurred through the function f_i, and likewise $\mathrm{Mults}(f_i)$ for field multiplications):

- Field additions: $3 + \mathrm{Adds}(f_1) + \mathrm{Adds}(f_2)$
- Field multiplications: $1 + \mathrm{Mults}(f_1) + \mathrm{Mults}(f_2)$
- Multiplicative depth: $\max\{\mathtt{D}(f_1), \max\{\mathtt{D}(r_{i-1}), \mathtt{D}(f_2)\} + 1\}$

It remains to analyze these parameters. The computation is split into three parts:

First, the effort for computing f_2 is computed in **Subsection 4.2**, then the effort for f_1 in **Subsection 4.3**, and lastly the results are combined for the total effort for each digit in **Subsection 4.4**. The effort for f_1 is again split into several parts: First, the effort for the closed formula from Theorem 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, whose analysis can be found in the appendix. This promises 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.

4.2 Effort for f_2

Recall from Theorem 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 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 $r_i = f_1(a_{i-1}, b_{i-1}) - r_{i-1} \cdot f_2(a_{i-1}, b_{i-1})$ 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:

- 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.
- To multiply $p-1$ factors, one needs $p-2$ multiplications in total.
- Implementing the multiplication in way with the least depth (i.e., in a fanned-out fashion[6]) we obtain a depth of $\lceil \log_2(p-1) \rceil$.

Thus, our effort for computing f_2 is:

- **Field additions:** $p-1$
- **Field multiplications:** $p-2$
- **Multiplicative depth:** $\lceil \log_2(p-1) \rceil$

4.3 Effort for f_1

4.3.1 Straightforward Approach. We first present the straightforward way of computation, and then show how to reduce the number of field multiplications in a time-memory tradeoff. 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

[6] 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 $\Big((a \cdot b) \cdot (c \cdot d)\Big) \cdot (e \cdot f)$ (depth 3) rather than $(((((a \cdot b) \cdot c) \cdot d) \cdot e) \cdot f)$ of depth 5.

(2) Compute $l_i(a), l_i(b)$ for $i = 1, \ldots, p-1$ using the precomputed factors from the previous step (see also the effort analysis from f_2):
- 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 $i = 2, \ldots, p-1$: $\sum_{j=1}^{i} l_{p-j}(a) = \left(\sum_{j=1}^{i-1} l_{p-j}(a) \right) + l_{p-i}(a)$, 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 $i = 1, \ldots, p-1$, which incurs one multiplication for each i and raises the multiplicative depth by one (remember, 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 $\sum_{i=1}^{p-1} \left(l_i(b) \cdot \sum_{j=1}^{i} l_{p-j}(a) \right)$:
- Field additions: $p-2$
- Field multiplications: 0
- Multiplicative depth: +0

In total, we obtain the effort as:

- Field additions: $4p-6$
- Field multiplications: $2p^2 - 5p + 3$
- Multiplicative depth: $\lceil \log_2(p-1) \rceil + 1$

4.3.2 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 $2p-4$. This example illustrates the idea behind the following time-memory tradeoff:

For convenience and readability, we will assume that all fractions are integers in our notation. In reality, if we are dividing a set with, for example, 17 elements, we will make one set with 8 elements and one with 9, but we will write $\frac{17}{2}$ in the following.

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 $\{x, (x-1), \ldots, (x-\frac{p}{2})\}$ and $\{(x-\frac{p}{2}+1), \ldots, (x-(p-1))\}$ and multiply the elements in each set to obtain two intermediate products[7] $L_2^1 := \prod_{j=0}^{\frac{p}{2}} (x-j)$ and $L_2^2 := \prod_{j=\frac{p}{2}+1}^{p-1} (x-j)$. 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 $\frac{p}{2}-1$ multiplications (because there the are $\frac{p}{2}-1$ missing factors and L_2^j that need to be multiplied). 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

$$L_4^1 := \prod_{j=0}^{\frac{p}{4}} (x-j), \ldots, L_4^4 := \prod_{j=\frac{3p}{4}+1}^{p-1} (x-j).$$

Each of these L_4^i can be computed with $\frac{p}{4}-1$ multiplications, 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 $l_i(x) = L_2^{j_1} \cdot L_4^{j_2} \cdot \tilde{r}$ for some $j_1 \in \{1,2\}, j_2 \in \{1,2,3,4\}$ and $\tilde{r}$ consists of $\frac{p}{4}-1$ 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 $2 \cdot (p-2+(p-1) \cdot \frac{p}{4}) = \frac{1}{2}p^2 + \frac{3}{2}p - 4$ 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 $\frac{p}{k}$ 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.

[7] 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: $L_k^i := \prod_{j=\frac{(i-1) \cdot p}{k}+1}^{\frac{i \cdot p}{k}} (x-j)$

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 $\frac{p}{k} - 1$ 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 to accomodate both a and b, we obtain $2 \cdot (p - 2 + (p-1) \cdot (\frac{p}{k} - 2 + \log_2(k))) = \frac{2}{k}p^2 + (2\log_2(k) - \frac{2k+2}{k}) \cdot p - 2 \cdot \log_2(k)$ 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 get $2 \cdot (2^{w-1} + 2^{w-2} + 2^{w-3} + \cdots + 2) = 2 \cdot ((\sum_{i=0}^{w-1} 2^i) - 1) = 2 \cdot (2^w - 1) - 1 \approx 2p - 4$.

Thus in total, we get as effort for computing f_1 with the closed formula:

- Field additions: $4p - 6$
- Field multiplications: $2p \cdot \log_2(p) - 2 \cdot \log_2(p) - 2 + p - 1$ $= 2p \cdot \log_2(p) + p - 2 \cdot \log_2(p) - 3$
- Multiplicative depth: $\lceil \log_2(p-1) \rceil + 1$

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.

4.3.3 Using the expanded polynomial. We note at this point that it seems as though 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 expected when the double sum is expanded. Concretely, the degree of the expanded polynomial seems to be p rather than the expected $2p - 2$. For the analysis of this expanded form of the polynomial, we refer the reader to Appendix B. 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. These results can be seen in Figure 2.

Thus, we use the field addition and multiplication metric from the closed formula, but the best possible depth of $\lceil \log_2(p) \rceil$ from the expanded formula in our effort analysis.

The effort for computing f_1 is bounded by:

- **Field additions:** $4p - 6$
- **Field multiplications:** $2p \cdot \log_2(p) + p - 2 \cdot \log_2(p) - 3$
- **Multiplicative depth:** $\lceil \log_2(p) \rceil$

4.4 Conclusion

Putting these numbers together with the effort for f_2 and our analysis from the beginning, 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, we derived a depth of $\max\{\mathsf{D}(f_1), \max\{\mathsf{D}(r_{i-1}), \mathsf{D}(f_2)\} + 1\}$. Using this formula for r_2, we get

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

From here on, it is clear that $\mathsf{D}(r_i) > \mathsf{D}(f_1) \geq \mathsf{D}(f_2)$, so the depth will increase by 1 with each i, leaving us with a total depth of $\mathsf{D}(r_i) = \lceil \log_2(p) \rceil + i - 1$.

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

- **Field additions:** $5p - 4$
- **Field multiplications:** $2p \cdot \log_2(p) + 2p - 2 \cdot \log_2(p) - 4$
- **Multiplicative depth:** $\lceil \log_2(p) \rceil + i - 1$

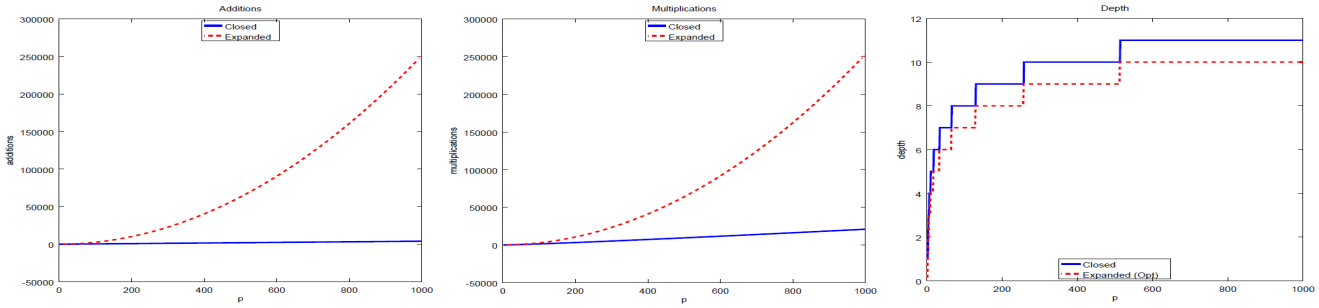
Special cases: The effort for computing $c_0 = a_0 + b_0$ is merely 1 field addition, and the cost for computing $c_1 = a_1 + b_1 + r_1 = a_1 + b_1 + f_1(a_0), b_0)$ is $4p + 4$ additions, $2p \cdot \log_2(p) + p - 2 \cdot \log_2(p) + 1$ multiplications, and a depth of $\lceil \log_2(p) \rceil$. Another special case is the most significant digit $c_{n+1} = r_{n+1}$, which has 2 field additions less than the other $c_i, i > 1$.

5 COST ANALYSIS FOR COMPUTING ON ENCRYPTED NATURAL NUMBERS

5.1 Overview

In this section, we analyze the effort required to add or multiply two natural numbers encoded p-adically using the polynomial we derived above.

In **Subsection 5.2**, we calculate the cost of adding two numbers in p-adic encoding. This involves determining the number of digits required for the respective base, and then using the costs per digit from the previous subsection to determine the total cost of addition. We also present **Theorem 2**, which tells us that $p = 2$ is the best choice in terms of field additions and multiplications. The proof of the theorem presents the asymptotic analysis, and we have graphed the costs for different numbers in p-adic encoding for all p under 1000. We also graph the multiplicative depth and see that the optimal choice of p for this metric depends on the size of the number being encoded, or more specifically, the required number of digits. Lastly, **Subsection 5.3** 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.



(a) Additions (b) Multiplications (c) Depth

Figure 2: Field additions, multiplications and (optimal) mult. depth to compute f_1 through the closed and expanded forms.

5.2 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 1 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 $4p - 5$ additions, $2p \cdot \log_2(p) + p - 2 \cdot \log_2(p) - 3$ multiplications and a depth of $\lceil \log_2(p) \rceil$.**

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. Next, we have $c_1 = a_1 + b_1 + r_1 = a_1 + b_1 + f_1(a_0, b_0)$ 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, $\ell > 2$, is:

- $9p - 9 + (\ell - 2) \cdot (5p - 4) = (5\ell - 1) \cdot p - (3\ell + 2)$ **field additions**
- $4p \cdot \log_2(p) + 3p - 4 \cdot \log_2(p) - 7 + (\ell - 2) \cdot (2p \cdot \log_2(p) + 2p - 2 \cdot \log_2(p) - 4) = 2\ell \cdot p \cdot \log_2(p) + (2\ell - 1) \cdot p - 2\ell \cdot \log_2(p) - 4 \cdot \ell + 1$ **field multiplications**
- **A multiplicative depth of $\lceil \log_2(p) \rceil + \ell - 1$.**

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 paper:

Theorem 2. *Using total number of additions or multiplications (or a balance between total number of multiplications and depth) as the cost metric, $p = 2$ is the most efficient encoding for adding two natural numbers in p-adic encoding.*

Proof. We can see from the above cases that while the required encoding length $\ell = \lfloor \log_p(x) \rfloor + 1 = \lfloor \frac{\log_2(x)}{\log_2(p)} \rfloor + 1$ only decreases logarithmically, the effort grows with p as $O(\ell \cdot p) = O((\lfloor \frac{\log_2(x)}{\log_2(p)} \rfloor + 1) \cdot p) \approx O(p + \frac{p}{\log_2(p)})$ (for additions) and as $O(\ell \cdot p \cdot \log_2(p)) = O((\lfloor \frac{\log_2(x)}{\log_2(p)} \rfloor + 1) \cdot p \cdot \log_2(p)) \approx O(p \cdot \log_2(p) + p)$ (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. □

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. Theorem 2 holds for p-adic encoding used in true FHE. We have illustrated this Theorem through Figure 1 on page 3, which shows the effort as p grows for selected values of x.

We can see that indeed, the number of additions, multiplications and the depth increase significantly as the encoding base p increases. 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 (which also occurs for the non-optimal depth $\lceil \log_2(p-1) \rceil + \ell$) 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 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 1c, the depth-optimal choices for adding x would be $p = 3$ for $x = 20$, $p = 7$ for $x = 7000$, and $p = 29$ for $x = 10^7$.

5.3 The Cost of Multiplying Two Natural numbers

In this section, we analyze the cost of multiplying two natural numbers $a_{\ell-1}a_{\ell-2} \ldots a_1 a_0 \cdot b_{\ell-1}b_{\ell-2} \ldots b_1 b_0$ with ℓ digits in p-adic encoding. In performing this multiplication, there are two main steps: First, we need to perform a one digit multiplication of each b_i

with all of $a_{\ell-1}a_{\ell-2}\dots a_1a_0$, shifting one space to the left with each increasing i. In the second step, we add up the rows we obtained in this way using the addition from the previous subsection as a building block. As an example, consider the multiplication of two 3-digit numbers:

a_2	a_1	a_0	$\cdot$	b_2	b_1	b_0
			x_3	x_2	x_1	x_0
		y_3	y_2	y_1	y_0	
	z_3	z_2	z_1	z_0		
	c_5	c_4	c_3	c_2	c_1	c_0

The first step is obtaining the rows $x_3x_2x_1x_0 = a_2a_1a_0 \cdot b_0$, $y_3y_2y_1y_0 = a_2a_1a_0 \cdot b_1$ and $z_3z_2x_1z_0 = a_2a_1a_0 \cdot b_2$. Except in the case of $p = 2$ (where $b_i \in \{0, 1\}$, so $x_3x_2x_1x_0 = (a_2 \cdot b_0)(a_1 \cdot b_0)(a_0 \cdot b_0)$), this actually requires some computational effort: In the case that $a_0 \cdot b_0 > p$, we have a carry into the next digit. Concretely, we write $a_i \cdot b_j + r_i = r_{i+1} \cdot p + x_i$ where $r_0 = 0$ and $r_i \in \{0, \dots, p-2\}$.[8] Very similarly to the uniqueness proof of Theorem 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. 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 that among these different possible polynomials, there seems to be one which has a degree of only $2 \cdot (p-1)+1$. Thus, in our effort analysis we will use this value as our lower bound on the depth, and a similar reasoning for number of additions and multiplications via the closed Lagrangian formula as in the previous section. Using precomputing as above, we obtain as effort for computing r_{i+1} from (a_i, b_i, r_i):

- Field additions: $6p - 9$
- Field multiplications: $3p \cdot \log_2(p) + 2p - 3 \cdot \log_2(p) - 5$
- Multiplicative depth: $\lceil \log_2(2p-1) \rceil$

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 and mind the special first and last digit):

- Field additions: $\ell \cdot (6p - 8) - 1$
- Field multiplications: $\ell \cdot (3p \cdot \log_2(p) + 2p - 3 \cdot \log_2(p) - 5)$
- Multiplicative depth: $\ell \cdot \lceil \log_2(2p-1) \rceil$

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

- Field additions: $\ell^2 \cdot (6p - 8) - 1$
- Field multiplications: $\ell^2 \cdot (3p \cdot \log_2(p) + 2p - 3 \cdot \log_2(p) - 5)$
- Multiplicative depth: $\ell \cdot \lceil \log_2(2p-1) \rceil$

The second step consists of adding all the rows that we computed in the first step. We apply the improvement presented in [22] where we copy the digits of the upper row over the blank spaces on the right to the result, and 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.

[8] 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 $r_i \leq p-2$.

First, we 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. Continuing this until we have only one row left (with $\texttt{Adds}(x)$ denoting the respective effort for adding two number of x digits as computed in the previous subsection), we get as total cost of this second step:

- Field additions: $\sum_{i=1}^{\lceil \log_2(\ell) \rceil} \frac{\ell}{2^i} \cdot \texttt{Adds}(\ell + 2^{i-1} + 1)$
- Field multiplications: $\sum_{i=1}^{\lceil \log_2(\ell) \rceil} \frac{\ell}{2^i} \cdot \texttt{Adds}(\ell + 2^{i-1} + 1)$
- Multiplicative depth: $\sum_{i=1}^{\lceil \log_2(\ell) \rceil} \texttt{Adds}(\ell + 2^{i-1} + 1)$

5.4 Conclusion

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

- **Field additions:** $\ell^2 \cdot (6p - 8) - 1 + \sum_{i=1}^{\lceil \log_2(\ell) \rceil} \frac{\ell}{2^i} \cdot \textbf{Adds}(\ell + 2^{i-1} + 1)$
- **Field multiplications:** $\ell^2 \cdot (3p \cdot \log_2(p) + 2p - 3 \cdot \log_2(p) - 5) + \sum_{i=1}^{\lceil \log_2(\ell) \rceil} \frac{\ell}{2^i} \cdot \textbf{Adds}(\ell + 2^{i-1} + 1)$
- **Multiplicative depth:** $\ell \cdot \lceil \log_2(2p-1) \rceil + \sum_{i=1}^{\lceil \log_2(\ell) \rceil} \textbf{Adds}(\ell + 2^{i-1} + 1)$

Since this is a rather complicated formula, we have plotted the costs for different inputs in Figure 3.

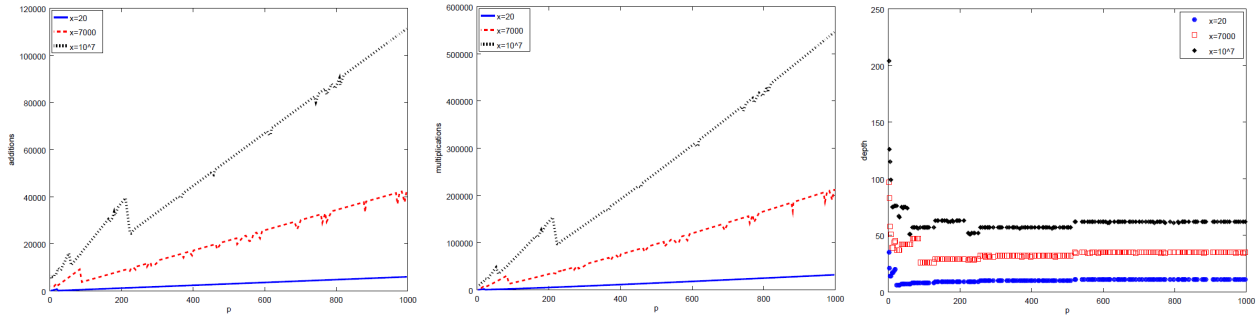
We can see that regarding additions and multiplications, the effort is lowest at $p = 2$ and grows with increasing p, though there are again some sharp drops when the required number of digits decreases. As one would expect, the issue with depth has propagated from addition, which we used as a building block in multiplication: The best depth for $x = 20$ would be $p = 23$, the best depth for $x = 7000$ would be $p = 89$, and the best depth for $x = 10^7$ would be $p = 59$. 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, depending on the specific outsourcing scenario.

6 USING $GF(P^K)$ AS ENCODING BASE

We now generalize our analysis tp arbitrary finite fields as encoding bases.

6.1 Analysis

Much in the same way as in Sections 4 and 5, we have also analyzed the effort incurred when using $GF(p^k)$ for a prime p and a



(a) Additions **(b) Multiplications** **(c) Depth**

Figure 3: Number of field additions, field multiplications and multiplicative depth for multiplying $x = 20/7000/10^7$ to a number of same size.

$k > 1$ as an encoding base. 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 $\mathbb{Z}_p$, through the insertion homomorphism: The element $a = \sum_{i=0}^{k-1} \alpha_i X^i \in GF(p^k)$ (with $\alpha_i \in \mathbb{Z}_p$) encodes the number $\tilde{a} = \sum_{i=0}^{k-1} \alpha_i p^i \in \mathbb{N}$. Then generalizing this to allow numbers larger than $p^k - 1$ is straightforward: We will represent a number a as $a_n a_{n-1} \dots a_1 a_0$ where $a_i \in GF(p^k)$ through $a = \sum_{j=0}^{n} \tilde{a_i}(p^k)^i$.

Example 1. *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: $8008 = 4096 + 3584 + 320 + 8 = 1\cdot 8^4 + 7\cdot 8^3 + 5\cdot 8^2 + 1\cdot 8^1 + 0\cdot 8^0 \mapsto 1\cdot 8^4 + (X^2 + X + 1)\cdot 8^3 + (X^2 + 1)\cdot 8^2 + 1\cdot 8^1 + 0\cdot 8^0$ which yields a tuple of $(1, X^2 + X + 1, X^2 + 1, 1, 0)$.*

Having determined this encoding, we 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 similar effort here and at least double the effort compared to $\mathbb{Z}_{p'}$ in total.

We now present the results of this analysis – the detailed computation can be found in Appendix C. To add two natural numbers, we have the following effort:

Case 1: $0 \le x \le p^k - 1$**:** This means that our number can be encoded with 1 digit, and the result will have two digits.

- Field additions: $2p^{2k} + 2p^k - 5$
- Field multiplications: $4p^k \cdot \log_2(p^k) + 2p^k - 2\log_2(p^k) - 7$
- Constant multiplications: p^{2k}
- Multiplicative depth: $\lceil \log_2(p^k - 1) \rceil + 1$

Case 2: $p^k \le x$**:** This means that x will be encoded with $2 \le \ell := \lfloor \log_p(x) \rfloor + 1$ digits and the result will have $\ell + 1$ digits.

- Field additions: $5p^{2k} + 8p^k - 12 + (\ell - 2)\cdot(3\cdot p^{2k} + 6p^k - 6)$ $= (3\ell - 1)\cdot p^{2k} + (6\ell - 4)\cdot p^k - 6\ell$
- Field multiplications: $10p^k \cdot \log_2(p^k) + 6p^k - 4\log_2(p^k) - 19 + (\ell - 2)\cdot(6p^k \cdot \log_2(p^k) + 4p^k - 2\log_2(p^k) - 11)$ $= (6\ell - 2)\cdot p^k \cdot \log_2(p^k) + (4\ell - 2)\cdot p^k - 2\ell \cdot \log_2(p^k) - 11\ell + 3$
- Constant multiplications: $3p^{2k} + 1 + (\ell - 2)\cdot(2p^{2k} + 1)$ $= (2\ell - 1)\cdot p^{2k} + \ell - 1$
- Multiplicative depth: $\lceil \log_2(p^k - 1) \rceil + \ell$

6.2 Conclusion

We now compare the calculated effort to:

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

Concretely, Figure 4 shows the effort of adding two numbers of the same size ($x = 20/7000/10^7$) in the respective encoding base for bases less than 1000. The blue points are the efforts for using $\mathbb{Z}_p$, the red squares for p^2, the black diamonds for p^3, and the green star groups all bases p^k with $k \ge 4$, since the primes p with $p^k \le 1000$ 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 $\mathbb{Z}_p$.

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. Recall from Section 5.2 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, and since we can choose a prime p' that

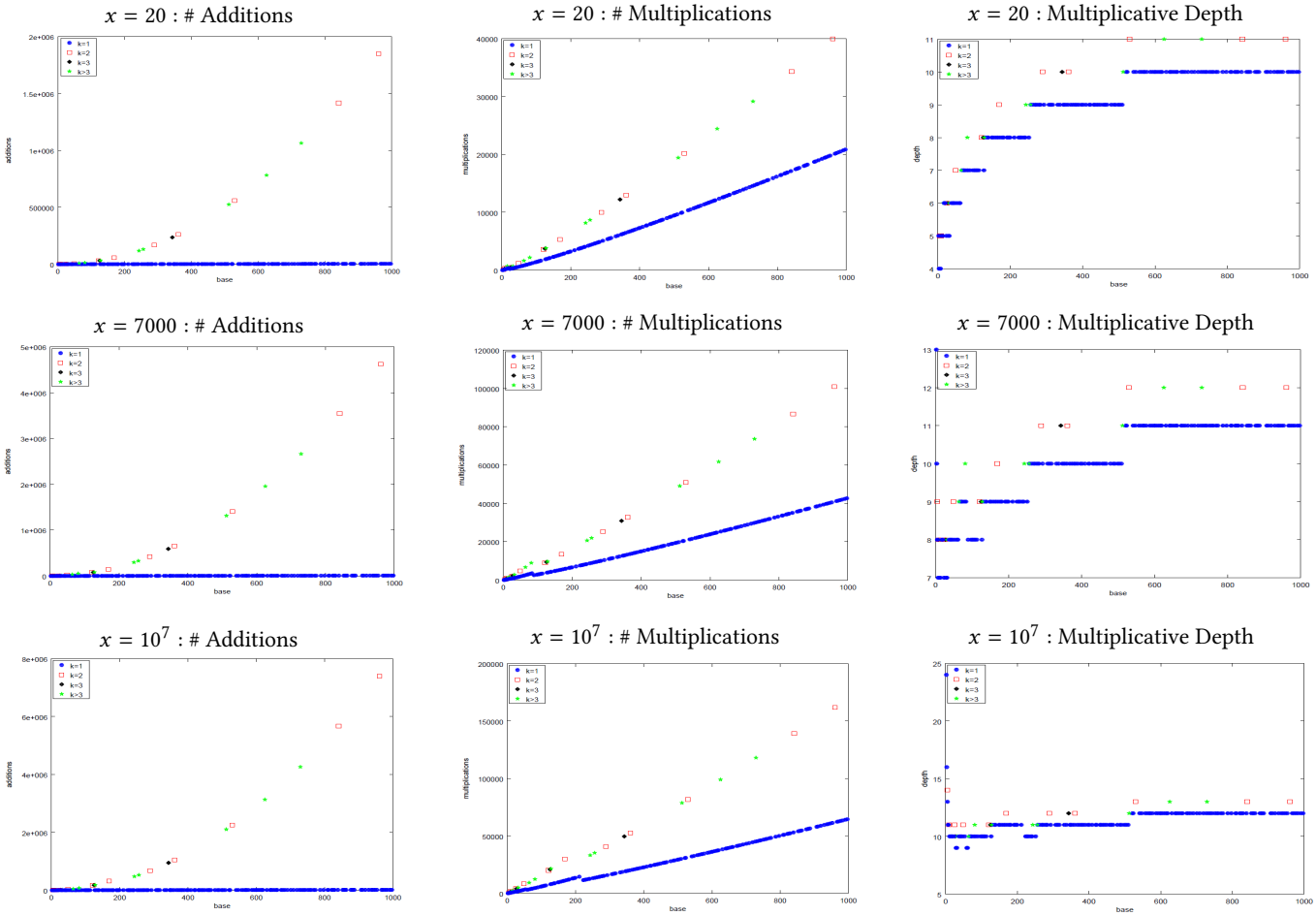


Figure 4: Number of field additions, field multiplications and multiplicative depth for multiplying $x = 20$ (first row)/$x = 7000$ (second row)/$x = 10^7$ (bottom row) to a number of same size for encoding base $GF(p^k)$.

is close to p^k (and thus requires roughly the same encoding length) which requires much less effort as shown in Figure 4, there is no case where choosing p^k as an encoding base with $k > 1$ brings any benefit. For this reason, p^k is always the worst encoding choices (and we are never without an alternative), so we do not continue with the analysis of $GF(p^k)$ as an encoding base.

7 RATIONAL NUMBERS AND INTEGERS

7.1 Overview

In most real-world applications, computing on natural numbers is not sufficient. In this section, we briefly discuss how to incorporate rational numbers of arbitrary but fixed precision (so that real numbers can be approximated) and at what cost (**Subsection 7.2**). Afterwards, we consider in **Subsection 7.3** (signed) integers by investigating the costs when using the encodings $p's$ Complements and Sign-Magnitude, respectively. It turns out that by switching between these two encodings, we can get the lowest cost as with only slightly higher effort than for natural numbers, in addition to the cost of switching (which is roughly that of one p-adic addition.). Thus, our results for natural numbers also extend to rational numbers, meaning that $p = 2$ is optimal with regard to field additions and multiplications, and depth does not have a generic optimum.

7.2 Representing Rational Numbers by Scaling

First, let the encoding base p be an arbitrary prime. Given a rational number that we wish to encode (and assuming for the moment that negative numbers are no problem), we need to transform this rational into an integer, which we can then encode p-adically in the next step. The probably most straightforward approach is to introduce a scaling factor. We explain that choosing a power of p as scaling factor is beneficial. That is, we pick a precision[9] σ with which we want to work with in the following. We now multiply the rational with p^σ and round or truncate to obtain an integer that we can encode.

The importance of choosing the scaling factor as a power of the encoding base is as follows: Suppose that we have two rational numbers A and B which we scale and round to $\tilde{A} = A \cdot p^\sigma$ and

[9] i.e., there are σ p-adic digits after the point.

$\tilde{B} = B \cdot p^\sigma$. After encoding p-adically and encrypting the individual digits, multiplying yields $\tilde{A} \cdot \tilde{B} = A \cdot B \cdot p^{2\sigma}$. Thus, after decrypting, the data owner needs to know what power of the scaling factor to divide the result by, leaking unwanted information about the function that was applied, which might be the computing party's secret. Also the required number of digits increases to accommodate the extra precision digits, making all computations less efficient. However, if we have a number in p-adic encoding, deleting the last σ digits corresponds to dividing by p^σ and truncating the result. This way, using p^σ as the scaling factor, the computing party can delete the σ least significant digits after each multiplication and thus keep the precision at a constant σ bits, increasing efficiency (by using less digits) and privacy (because the data owner now divides the result by p^σ regardless of the function that was applied).

7.3 Encoding Integers

Now that we have seen how to transform rationals into integers and work with them efficiently, we need to look into incorporating negative numbers into our p-adic encoding from the previous sections. Generalizing from $p = 2$, for which these encodings are well known, we investigate two main approaches: p's Complement and Sign-Magnitude. Note that this question has been extensively studied in [22] but for the case $p = 2$ only. In the following, we explain shortly how the results extend to the case of $p > 2$. As the extension is quite straightforward, we omit the technical details and only sketch the main arguments.

7.3.1 p's Complement. In p's Complement encoding, elements have the form $a_n \dots a_1 a_0$ with $a_i \in \{0, 1, \dots, p-1\}$ for $i = 0, \dots, n-1$, and $a_n \in \{0, 1\}$, where $x = -a_n \cdot p^n + \sum_{i=0}^{n-1} a_i \cdot p^i$. This means that the first digit encodes either 0 or $-p^n$ and the following digits correspond to the "normal" p-adic encoding. As an example, suppose $p = 3$. Then 0201 evaluates to $0 \cdot (-3^3) + 2 \cdot 3^2 + 0 \cdot 3 + 1 = 19$, whereas $-19 = -27 + 8 = -3^3 + 0 \cdot 3^2 + 2 \cdot 3 + 2$ would be encoded as 1022. In the case that $p = 2$, this is the encoding that is usually used in the internal workings of a computer.

Addition. Addition works in much the same way as normal p-adic addition, except for one point: To obtain the correct result when adding two n-digit numbers, the result must also be encodeable by n digits, and any values past the n^{th} digit are discarded. Since the result of adding two n-digit numbers is usually $n + 1$ digits long, we first extend the inputs by one digit without changing their value so that we can then add two $n + 1$ digit numbers whose sum is also encodeable by $n + 1$ digits, yielding the correct result. Note that to extend the number of digits by k, we must merely insert k 0's (for a positive number) or $p - 1$'s behind the most significant digit. This is called *sign extension*. However, the most significant digit may come out wrong, as the result will be mod p though in reality, it is mod 2. This can be fixed by replacing c_n with $\Big(c_n \cdot (c_n - 2) \cdot (c_n - 4) \dots (c_n - (p-1))\Big)^{p-1}$. In its expanded form, this adds a depth of $\lceil \log_2(p-1) \rceil$, and incurs $\frac{p-1}{2}$ additions and $\frac{p-1}{2} + p - 2$ multiplications in its closed form, which is less than a full addition of natural numbers. Obviously, this is not necessary if $p = 2$, so this further increases the efficiency of $p = 2$ compared to other p.

Conclusion: This means that the effort of adding two numbers in p's Complement encoding is comparable to twice the effort for adding two natural numbers derived in Section 5.2, except that the depth is twice as large. For $p = 2$, it is almost exactly the same effort as adding two natural numbers in binary encoding.

Multiplication. When multiplying two numbers in p's Complement encoding, we need to follow a few steps. For maximum generality, we assume that the input numbers have m and n digits, respectively.

(1) Increase the the number of digits for both numbers through sign extension (as described above) to length $m + n$.
(2) Perform regular p-adic 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 $n + m$ digits.

Because of the sign extension required in the first step, not only are the rows longer than for natural number multiplication ($n + m$ as compared to n), but there are also more of them ($n+m$ as opposed to m), so we must do more additions with inputs of greater lengths.

Conclusion: This means that p's Complement roughly requires the same multiplication effort as a natural number with twice as many digits, i.e., multiplying a natural number x with k digits in p's Complement encoding requires roughly the same effort as multiplying x^2 (which has $2k$ digits) in "regular" p-adic encoding.

7.3.2 Sign-Magnitude. The second encoding that we consider, Sign-Magnitude, is the most obvious approach to incorporating negative numbers: The absolute value of the number is encoded p-adically as a natural number, and there is an extra digit (the most significant digit) which determines the sign. Concretely, elements in this encoding have the form $a_n a_{n-1} \dots a_1 a_0$ with $a_i \in \{0, 1, \dots, p-1\}$ for $i = 0, \dots, n-1$, and $a_n \in \{0, 1\}$, where $x = (-1)^{a_n} \cdot \sum_{i=0}^{n-1} a_i \cdot p^i$. It is easy to see that for positive numbers, this encoding is the same as p's complement. Continuing the example from above where $p = 3$, we again evaluate 0201 as $(-1)^0 \cdot (2 \cdot 3^2 + 0 \cdot 3 + 1) = 19$. However, $-19 = (-1)^1 \cdot (2 \cdot 3^2 + 0 \cdot 3 + 1)$ is now encoded as 1201. As we can easily see, a numbers is changed to its negative simply by changing the first digit. This encoding suffers from having two representations of 0: $00 \dots 0$ and $100 \dots 0$.

Addition. Adding two numbers in Sign-Magnitude representation is surprisingly complex, at least when the data is encrypted[10]. Concretely, if the two most significant digits are equal, one drops them, adds the remaining digits using the addition for natural numbers, and add the most significant bit to the front again. However, if they are unequal, we must compare the absolute values, subtract (in a routine which we have not defined in this paper - it has effort roughly like addition) the smaller from the larger, and keep the sign of the larger value. As the reader may have noticed, we also need a comparison function, which in itself has several different case branches which we must all compute. Using the extended version of [22] (in which the comparison function is derived in the appendix)

[10]When computing in the clear, one only has to compute the appropriate branch when a computation splits up via `IF`-clauses. When computing on encrypted data, however, one cannot see what the appropriate branch is, and thus has to compute all possibilities, multiplying each with a (encrypted) boolean variable expressing whether that branch is true and adding the results.

as our basis, we estimate the cost of comparing two numbers as more than three times the cost of adding those numbers.

Conclusion: The total cost of adding two numbers in Sign-Magnitude encoding is roughly one p-adic addition, two comparisons (each having costs of about three additions) and 4 subtractions (where subtraction and addition have about the same cost). This yields already 11 additions in "regular" p-adic encoding, being significantly more costly than using $p's$ Complement encoding. Note that the effort induced by choosing the correct branch is not included yet, meaning that the total effort will be even higher.

Multiplication. Multiplication with Sign-Magnitude encoding is conceptually very simple: We delete the sign digits a_n and b_n, multiply the remaining digits as with regular p-adic multiplication, and append the correct sign digit c_n. This last part is easy: Since the sign digits $a_n, b_n \in \{0, 1\}$, the function $c_n = (a_n + b_n) \cdot (a_n + b_n - 2) \cdot (p-1)$ computes the correct value: 0 when $a_n = b_n$, and 1 otherwise. To add the individual rows that we obtain during multiplication, we do not have to use the complicated addition as presented above, but rather the much more efficient addition of natural p-adic numbers, as we have discarded the sign during multiplication.

Conclusion: The effort of multiplying two numbers in Sign-Magnitude encoding is roughly the same as multiplying them with "regular" p-adic encoding.

7.3.3 Hybrid Encoding. We see that the choice of encoding to incorporate negative integer numbers can make a big difference in performance. Since p's Complement addition is more efficient than Sign-Magnitude, but Sign-Magnitude is more efficient for multiplication, we believe that a hybrid approach like in [22] would also be the best choice here: One does all additions in p's Complement encoding, and for multiplication switches the encoding to Sign Magnitude. Using this, one can have roughly the same operation cost as for natural numbers in p-adic encoding (slightly more for additions), plus the cost of switching between encodings, which is roughly that of one p-adic addition. Of course, since this already holds true for natural numbers in p-adic encoding, the choice $p = 2$ by far incurs the least amount of field additions and multiplications in these two encodings and the hybrid encoding also, while the optimal depth choice remains variable.

8 RELATED WORK

While encryption schemes that allow one type of operation on ciphertexts are well understood and have a comprehensive security characterization [3], Fully Homomorphic Encryption, which allows both unlimited additions and multiplications, was only first solved in [17]. Since then, numerous other schemes have been developed, for example [27] (the most approachable in its simplicity), [6] (currently considered the most efficient one), and several others like [26], [11], [12], [16] and [19]. An overview can be found in [2]. These schemes usually have a plaintext space of $\mathbb{Z}_p[X]/(F(x))$ where $F(x)$ is a cyclotomic polynomial, though [27] was first defined over $\mathbb{Z}_2$ and extended to other plaintext spaces in [25], and [15] is a notable exception with a plaintext space of $\{0, 1\}$ using the NAND operator.

There are several libraries implementing FHE, with [21] usually considered as the fastest. Numerous works have been published concerning actual implementation of FHE, like [18] (homomorphically evaluating the AES circuit), [5] (predictive analysis on encrypted medical data), or [20] (machine learning on encrypted data), and [24] discusses whether FHE will ever be practical and gives a number of possible applications. Most of these applications work by embedding their computations into a very large plaintext space and using SHE, as described in the introduction.

The recent increase in papers regarding encoding for FHE illustrates its importance: [10] examines encoding rational numbers through continued fractions (restricted to positive rationals and evaluating linear multivariate polynomials), whereas [13] focuses on most efficiently embedding the computation into a single large plaintext space. Another work that explores similar ideas as [13] and also offers an implementation is [14]. An extension of [16] allowing floating point numbers is presented in [1], and [8] gives a high-level overview of arithmetic methods for FHE, but resticted to positive numbers. In [22], arithmetic operations and different binary encodings for rational numbers are examined and compared in their effort. [4] explores a non-integral base encoding, and [28] (seemingly having significant overlap with [22]), presents different arithmetic algorithms including a costly division, though apparently limited to positive numbers. Lastly, [9] allows approximate operations by utilizing noise from the encryption itself. To our knowledge, there are no papers concerned with the costs of encoding in a base other than $p = 2$ except [23], which exclusively analyzes [25] and uses different cost metrics. 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 [27], and does not include an effort analysis.

9 CONCLUSION AND FUTURE WORK

In conclusion, we have shown that among all plaintext spaces of the form $GF(p^k)$, the choice $\mathbb{Z}_2$ is optimal with regard to the number of field operations when computing on encrypted numbers. For the cost metric of multiplicative depth, there is no generic optimum, as the optimum depends heavily on the circumstances – however, choosing $k > 1$ is never a good choice. We show how to extend p-adic encoding of natural numbers to rationals and see that the above results also hold in this case.

For future work, we intend to extend our analysis to other representations like the Non-Adjacent Form and Redundant Digit Encoding, where the latter seems very promising as it would allow addition without having to propagate the carry.

REFERENCES

[1] Seiko Arita and Shota Nakasato. 2016/402. Fully Homomorphic Encryption for Point Numbers. *IACR Cryptology ePrint Archive* (2016/402).

[2] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A. Reuter, and Martin Strand. 2015/1192. A Guide to Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive* (2015/1192).

[3] Frederik Armknecht, Stefan Katzenbeisser, and Andreas Peter. 2013. Group homomorphic encryption: characterizations, impossibility results, and applications. *DCC* (2013).

[4] 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* (????).

[5] Joppe W. Bos, Kristin E. Lauter, and Michael Naehrig. 2014. Private predictive analysis on encrypted medical data. *Journal of Biomedical Informatics* 50 (2014).

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2011. Fully Homomorphic Encryption without Bootstrapping. *ECCC* 18 (2011).
[7] Hao Chen, Kim Laine, and Rachel Player. 2017. Simple Encrypted Arithmetic Library - SEAL v2.1. *IACR Cryptology ePrint Archive* 2017 (2017), 224.
[8] Yao Chen and Guang Gong. 2015. Integer arithmetic over ciphertext and homomorphic data aggregation. In *CNS*.
[9] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2016/421. Homomorphic Encryption for Arithmetic of Approximate Numbers. *IACR Cryptology ePrint Archive* (2016/421).
[10] HeeWon Chung and Myungsun Kim. 2016/344. Encoding Rational Numbers for FHE-based Applications. *IACR Cryptology ePrint Archive* (2016/344).
[11] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. 2014. Scale-Invariant Fully Homomorphic Encryption over the Integers. In *PKC*.
[12] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. 2012. Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers. In *EUROCRYPT*.
[13] Anamaria Costache, Nigel P. Smart, S. Vivek, and A. Waller. 2016/250. Fixed Point Arithmetic in SHE Scheme. *IACR Cryptology ePrint Archive* (2016/250).
[14] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2015. *Manual for Using Homomorphic Encryption for Bioinformatics.* Technical Report MSR-TR-2015-87. Microsoft Research.
[15] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *EUROCRYPT*.
[16] Junfeng Fan and Frederik Vercauteren. 2012/144. Somewhat Practical Fully Homomorphic Encryption. *IACR Cryptology ePrint Archive* (2012/144).
[17] Craig Gentry. 2009. *A fully homomorphic encryption scheme.* Ph.D. Dissertation. Stanford University.
[18] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Homomorphic Evaluation of the AES Circuit. In *CRYPTO*.
[19] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *CRYPTO*.
[20] Thore Graepel, Kristin E. Lauter, and Michael Naehrig. 2012. ML Confidential: Machine Learning on Encrypted Data. In *ICISC*.
[21] Shai Halevi and Victor Shoup. 2015. HeLib Library. (2015). https://github.com/shaih/HElib
[22] Angela Jäschke and Frederik Armknecht. 2016. Accelerating Homomorphic Computations on Rational Numbers. In *ACNS*.
[23] Eunkyung Kim and Mehdi Tibouchi. 2016. FHE Over the Integers and Modular Arithmetic Circuits. In *CANS*. 435–450.
[24] Michael Naehrig, Kristin E. Lauter, and Vinod Vaikuntanathan. 2011. Can homomorphic encryption be practical?. In *CCSW*.
[25] Koji Nuida and Kaoru Kurosawa. 2015. (Batch) Fully Homomorphic Encryption over Integers for Non-Binary Message Spaces. In *EUROCRYPT*.
[26] Nigel P. Smart and Frederik Vercauteren. 2010. Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes. In *PKC*.
[27] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. 2010. Fully Homomorphic Encryption over the Integers. In *EUROCRYPT*.
[28] Chen Xu, Jingwei Chen, Wenyuan Wu, and Yong Feng. 2016. Homomorphically Encrypted Arithmetic Operations Over the Integer Ring. In *ISPEC*.

A PROOF OF THEOREM 1

This Section contains the proof of Theorem 1 from Section 3. 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 1**, which derives from Lemma 2, showing that the involved functions are symmetric. We then present **Lemmata 3 and 4**, which will be needed in later proofs. Lastly, **Theorem 1** presents the main result of this section, namely the closed formula for r_i.

Recall that our goal is to find the polynomial that expresses the carry bit as a function of the inputs and carry from the previous position: $r_i(a_{i-1}, b_{i-1}, r_{i-1})$.

Lemma 1. *The carry r_i is at most 1 for all i.*

Proof. We prove this by induction over the position $i \in \{0, \ldots, n+1\}$:
$i = 0$: This is the first position and thus there is no carry from a previous position, i.e., $r_0 = 0 \leq 1$.

Now for a general position $i > 0$, suppose it holds that $r_{i-1} \leq 1$. 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, we need to carry less than 2, i.e., at most 1, over to the next position (which corresponds to the next higher power of p). Thus, the next carry $r_i \leq 1$. □

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

$$r_i = \begin{cases} 0, & a_{i-1} + b_{i-1} + r_{i-1} \leq p-1 \\ 1, & a_{i-1} + b_{i-1} + r_{i-1} \geq p \end{cases} \tag{3}$$

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

Lemma 2. *The polynomial for computing $r_i(a_{i-1}, b_{i-1}, r_{i-1})$ has the form*

$$r_i = f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1}) \tag{4}$$

where f_1, f_2 are polynomials in two variables with

$$f_1(a,b) = \begin{cases} 0, & a+b \leq p-1 \\ 1, & a+b \geq p \end{cases} \quad \textit{and} \quad f_2(a,b) = \begin{cases} 1, & a+b = p-1 \\ 0, & \textit{else} \end{cases} \tag{5}$$

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 $r_i = f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1})$ 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 r_{i-1} is 0, so it must hold that

$$f_1(a,b) = \begin{cases} 0, & a+b \leq p-1 \\ 1, & a+b \geq p. \end{cases} \tag{6}$$

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 6, 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:

$$f_2(a,b) = r_i - f_1(a,b) = \begin{cases} 1, & a+b = p-1 \\ 0, & \text{else.} \end{cases} \tag{7}$$

□

Corollary 1. *Both $f_1(a,b)$ and $f_2(a,b)$ are symmetric*[11]*. By extension, $r_i(a,b,r_{i-1})$ itself is also symmetric in a and b, i.e., $r_i(a,b,r_{i-1}) = r_i(b,a,r_{i-1})$.*

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

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

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

□

The following Lemma will aid us in the proof of our main theorem:

Lemma 3. *For all $i \in \{0,...,p-1\}$, it holds that*

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

Proof. We rearrange the product in the following way:

$$\prod_{\substack{j=0\\ j\neq i}}^{p-1} \frac{1}{i-j} = \frac{1}{i}\cdot\frac{1}{i-1}\cdot\ldots\cdot\frac{1}{i-(i-1)}\cdot\frac{1}{i-(i+1)}\cdot\ldots\cdot\frac{1}{i-(p-1)}$$

$$=\frac{1}{i}\cdot\frac{1}{i-1}\cdot\ldots\cdot\frac{1}{1}\cdot\frac{1}{p-1}\cdot\ldots\cdot\frac{1}{i+1} = \prod_{j=1}^{p-1}\frac{1}{j}.$$

Since $x \mapsto x^{-1}$ is a bijection on $\mathbb{Z}_p^* = \{1,\ldots,p-1\}$,we have:

$$\prod_{j=1}^{p-1}\frac{1}{j} = \prod_{j=1}^{p-1} j = (p-1)! \tag{8}$$

Note that $(p-1)!$ is a multiplication of all invertible elements of $\mathbb{Z}_p$. Also, the equation $x^2 = 1$ has exactly two roots over $\mathbb{Z}_p$: 1 and $p-1$. Thus, for all elements $x \in \mathbb{Z}_p^*, x \notin \{1, p-1\}$ (i.e., for all $x \in \{2,\ldots,p-2\}$), it holds that $x \neq x^{-1}$. This means that $(p-1)!$ contains the inverse of every element except 1 and $p-1$. Thus, all elements except $p-1$ "cancel out" by being multiplied with their inverse, so we get

$$(p-1)! = p-1 \bmod p. \tag{9}$$

Thus, in total, we have

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

□

Recall that we defined

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

We prove the following attribute:

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

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

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 Supplementary Lemma 3, 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 1. *The formula for computing $r_i(a_{i-1}, b_{i-1}, r_{i-1})$ is*

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

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

$$f_1(a,b) = \sum_{i=1}^{p-1}\left(l_i(b)\cdot\sum_{j=1}^{i} l_{p-j}(a)\right) \tag{10}$$

and

$$f_2(a,b) = (p-1)\cdot l_{p-1}(a+b). \tag{11}$$

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

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

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 4, all $l_i(\tilde{b})$ with $i \neq \tilde{b}$ evaluate to zero. Since both sums involved in Equation 10 start at 1 rather than 0, the sums evaluate to 0 if $\tilde{a} = 0$ (then the terms of the inside sum 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 4). We can thus rewrite the right part of Equation 10 as

$$\sum_{i=1}^{p-1}\Big(l_i(\tilde{b})\cdot\sum_{j=1}^{i} l_{p-j}(\tilde{a})\Big) = l_{\tilde{b}}(\tilde{b})\cdot\sum_{j=1}^{\tilde{b}} l_{p-j}(\tilde{a}) = (p-1)\cdot\sum_{j=1}^{\tilde{b}} l_{p-j}(\tilde{a}). \tag{12}$$

Now we distinguish two cases:

Case 1: $\tilde{a}+\tilde{b} \le p-1$, which means that $f_1(\tilde{a},\tilde{b}) = 0$ according to Lemma 2. We also have

$$\tilde{a}+\tilde{b} \le p-1 \Leftrightarrow \tilde{a} \le p-\tilde{b}-1. \tag{13}$$

Looking at the inner sum of Equation 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 13), $l_{\tilde{a}}(\tilde{a})$ is not included in this sum, and thus all terms of the inner sum evaluate to 0 (Lemma 4), making the whole sum 0 when $\tilde{a}+\tilde{b} \le p-1$:

$$\sum_{i=1}^{p-1}\Big(l_i(\tilde{b})\cdot\sum_{j=1}^{i} l_{p-j}(\tilde{a})\Big) = l_{\tilde{b}}(\tilde{b})\cdot\sum_{j=1}^{\tilde{b}} l_{p-j}(\tilde{a}) = (p-1)\cdot\sum_{j=1}^{\tilde{b}} 0 = (p-1)\cdot 0 = 0. \tag{14}$$

Case 2: $\tilde{a}+\tilde{b} \ge p$, which means that $f_1(\tilde{a},\tilde{b}) = 1$ according to Lemma 2. By the same argument as in Case 1, we have

$$\tilde{a}+\tilde{b} \ge p \Leftrightarrow \tilde{a} \ge p-\tilde{b}. \tag{15}$$

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 4. Combining this with Equation 12, the sum now evaluates to

$$\begin{aligned}\sum_{i=1}^{p-1}\Big(l_i(\tilde{b})\cdot\sum_{j=1}^{i} l_{p-j}(\tilde{a})\Big) &= l_{\tilde{b}}(\tilde{b})\cdot\sum_{j=1}^{\tilde{b}} l_{p-j}(\tilde{a}) = (p-1)\cdot\sum_{j=1}^{\tilde{b}} l_{p-j}(\tilde{a}) \\ &= (p-1)\cdot l_{\tilde{a}}(\tilde{a}) = (p-1)\cdot(p-1) = 1\end{aligned} \tag{16}$$

where computations are mod p.

Combining equations 14 and 16, we get:

$$\sum_{i=1}^{p-1}\Big(l_i(\tilde{b})\cdot\sum_{j=1}^{i} l_{p-j}(\tilde{a})\Big) = \begin{cases}0, & a+b \le p-1 \text{ (Case 1)}\\ 1, & a+b \ge p \text{ (Case 2)}\end{cases} = f_1(a,b). \tag{17}$$

Uniqueness: To prove uniqueness, we take quick look at how our polynomial was derived by recalling Lagrange's polynomial interpolation[12]. The idea 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 $f_b = f_1|_b(a)$:

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

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 $f_1(a,b)$:

$$f_1(a,b) = \sum_{i=0}^{p-1} h_i(b)\cdot f_i(a) = \sum_{i=0}^{p-1} h_i(b)\cdot\Big(\sum_{j=0}^{p-1} h_j(a)\cdot f(j,i)\Big). \tag{18}$$

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

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

Using Lemma 3, we see that

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

Now we can rewrite Equation 18 as

$$\begin{aligned} f_1(a,b) &= \sum_{i=0}^{p-1}(p-1)\cdot l_i(b)\cdot\Big(\sum_{j=0}^{p-1}(p-1)\cdot l_j(a)\cdot f(j,i)\Big) \\ &= (p-1)^2\cdot\sum_{i=0}^{p-1} l_i(b)\cdot\sum_{j=0}^{p-1} l_j(a)\cdot f(j,i) \\ &= \sum_{i=0}^{p-1} l_i(b)\cdot\sum_{j=0}^{p-1} l_j(a)\cdot f(j,i).\end{aligned} \tag{19}$$

Lastly, since $f(j,i) = \begin{cases}1, & i+j \ge p\\ 0, & \text{else}\end{cases}$, we see that only $l_j(a)$ with $i+j \ge p \Leftrightarrow j \ge p-i \Leftrightarrow j \in \{p-i,\ldots,p-1\}$ 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 facts:

(1) As a polynomial ring over a field, $\mathbb{Z}_p[a]$ is a factorial ring.
(2) As a polynomial ring over a factorial ring, $(\mathbb{Z}_p[a])[b] \cong \mathbb{Z}_p[a,b]$ is a factorial ring.
(3) In a factorial ring (or more generally, an integral domain), a polynomial of degree $n \ge 1$ 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 $b \in \{0,\ldots,p-1\}$ be fixed but arbitrary and consider the polynomial $f_b(a)$, which was derived through Lagrangian interpolation in the points (a,b) for all $a \in \{0,\ldots,p-1\}$ as described above. Now assume that there is a different polynomial $g(a) \ne f_b(a)$ of equal or less degree (which is $p-1$) with $g(a) = f_b(a) = f_1(a,b)$ for all $a \in \{0,\ldots,p-1\}$. 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 $a = 0,\ldots,p-1$).

[12] Given $k+1$ points $(x_i, y_i = f(x_i))$, $i = 0,\ldots,k$, the Lagrangian interpolation polynomial (which interpolates the function f through these points) is given as

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

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 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 $r_i(a_{i-1}, b_{i-1}, r_{i-1})$. □

B ANALYSIS OF THE EXPANDED FORMULA

In this appendix, we present some results and the effort analysis regarding the expanded form of f_1.

Lemma 5. *In the expanded form of $l_i(x)$, the term x^k has the coefficient $i^{p-(k+1)} \bmod p$ where $1 \le k \le p-1$ and $i \in \{1,\dots,p-1\}$. 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

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

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

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

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 2. *The coefficient of the term $a^x b^y$ in the polynomial f_1 is*

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

Proof. We first recall Formula 10:

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

Using Lemma 5, 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 $x,y \in \{1,\dots,p-1\}$) will have the coefficient $\sum_{i=1}^{p-1}\left(i^{p-y-1}\cdot\sum_{j=1}^{i}(p-j)^{p-x-1}\right)$. Since the order in the exponent mod p is $p-1$, we can equivalently write $\sum_{i=1}^{p-1}\left(i^{-y}\cdot\sum_{j=1}^{i}(p-j)^{-x}\right)$. 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 $a^x\cdot b^y$:

$$(p-1)^{-x}\cdot\sum_{i=1}^{p-1}\left(i^{-y}\cdot\sum_{j=1}^{i} j^{-x}\right). \tag{20}$$

□

Corollary 3. *f_1 has degree at least p.*

Proof. From Corollary 2, we know that the coefficient of the term $a^{p-1}\cdot b$ is

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

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

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

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. □

Conjecture 1. *f_1 has exactly degree p.*

Intuition: This was the case for all p that we tested, and it was possible for us to show that the coefficient $(p-1)^{-x}\cdot\sum_{i=1}^{p-1}\left(i^{-y}\cdot\sum_{j=1}^{i} j^{-x}\right)$ is 0 for $p+1 \le x+y \le 2p-2$ when the parity of x and y is equal. However, the other case proves to be elusive, so we leave this statement as a conjecture. Luckily, as we will show in the following, this is not very important because the closed formula will turn out to be the better choice anyway.

Nonetheless, we present this expanded form of f_1 for the first few primes[13]:

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

As noted above in Conjecture 1, the polynomial in its expanded form seems to have a degree of only p instead of the expected $2p-2$, implying a theoretical best depth of $\lceil \log_2(p) \rceil$. 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 (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 dies 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: maximum $\lceil \log_2(p-1) \rceil$

(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 5.

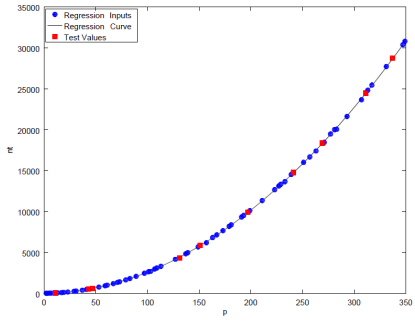


Figure 5: The number of terms for different p, the regression curve from these terms, and the test set.

As can easily be seen, the curve fits the data extremely well - the values predicted by this formula compared to the actual values are shown in Table 1.

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

- **Field additions:** $0.25p^2 + 0.87p + 2.15$
- **Field multiplications:** $0.25p^2 + 2.87p - 0.85$
- **Constant multiplications:** $0.25p^2 + 0.87p + 3.15$
- **Multiplicative depth:** $\lceil \log_2(p-1) \rceil + 1$**, theoretical best:** $\lceil \log_2(p) \rceil$

We can see that as p increases, the closed formula has much lower computation effort. Concretely, the number of operations to compute f_1 can be seen in Figure 6.

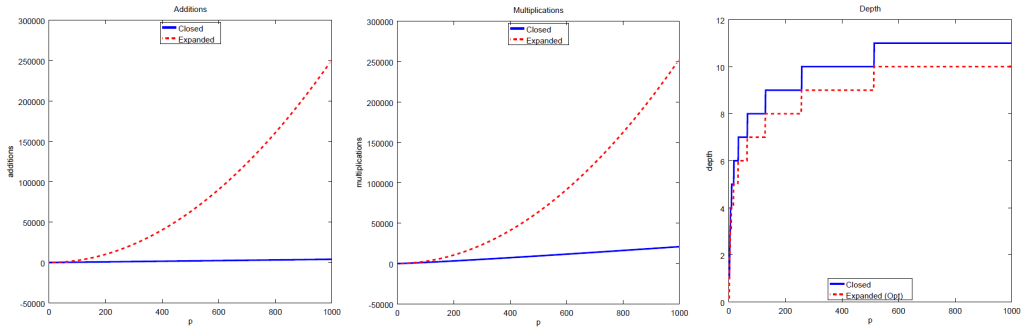
In addition, we would like to point out that if one were to implement this p-adic encoding, the closed formula can easily be realized by a loop, whereas it is questionable if one would actually want to implement the expanded form with e.g. nearly 10000 terms for $p = 197$.

To be an unbiased as possible, we took the better of the two values as an upper bound on the effort.

[13] The code to generate these polynomials for any p is available upon request.

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

Table 1: Actual vs. estimated values for the number of terms in the expanded form of f_1 for the values of the test set.



(a) Additions **(b) Multiplications** **(c) Depth**

Figure 6: 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).

C DETAILED ANALYSIS FOR $GF(P^K)$ AS ENCODING BASE

In this section, we analyze the situation for using $GF(p^k)$ for a prime p and a $k > 1$ as an encoding base, with the encoding as described in Section 6.

C.1 Effort for c_i

As mentioned previously, in this encoding we cannot simply write $c_i = a_i + b_i + r_i$ because the field addition is not the correct operation. Instead, we 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. Concretely, the two functions are detailed in the following.

C.1.1 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 $\mathbb{N}$)

$$f_1(a,b) = \begin{cases} 0, & a + b \leq p^k - 1 \\ 1, & a + b \geq p^k \end{cases} \text{ and } f_2(a,b) = \begin{cases} 1, & a + b = p^k - 1 \\ 0, & \text{else} \end{cases} \tag{21}$$

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 (not counting the multiplication with $p - 1$ because it is a constant) and $p^k - 1$ additions. Thus, for f_2 we have an effort of

- **Field additions:** $2p^k - 1$
- **Field multiplications:** $p^k - 2$
- **Constant multiplications:** 1
- **Multiplicative depth:** $\lceil \log_2(p^k - 1) \rceil$.

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

$$f_1(a,b) = \sum_{j \in GF(p^k)^*} \left(l_j(b) \cdot \sum_{i \in GF(p^k)^*} l_i(a) \cdot \tilde{r}(i,j) \right) \tag{22}$$

where

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

is supplied as a known value in Lagrangian interpolation[14]. We call the above Formula 22 the closed formula.

Note:. *We only sum over the non-zero elements of $GF(p^k)$ because if either i or j is zero, 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 $\mathbb{Z}_p$. However, due to the more abstract nature of $GF(p^k)$ and the non-trivial mapping to $\mathbb{N}$, 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*

[14]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.

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

$p\backslash k$	**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: Actual degree of f_1 vs. $2p^k - 2$

$p\backslash k$	**1**	**2**	**3**	**4**	**5**
2	1	4	10	22	46
3	1	9	39	129	399
5	1	25	205	1105	
7	1	49	595		
11	1	121			

$p\backslash k$	**1**	**2**	**3**	**4**
2	0	2	6	14
3	0	6	24	78
5	0	20	120	620
7	0	42	336	

Table 3: Degrees of g_1 and g_2.

a multiplication since it is always either 0 *or* 1*, so we always either add the term or don't.*

In reality, we again have the effect that the expanded formula (obtained by multiplying out the closed formula) seems to have a lower degree. Concretely, Table 2 shows the actual degree of f_1 for different values of p and k and the expected value of $2 \cdot p^k - 2$.

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 4.3, 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 4.3, 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 $2 \cdot \left(p^k - 2 + (p^k - 1) \cdot (\log_2(p^k) - 1)\right) = 2p^k \cdot \log_2(p^k) - 2\log_2(p^k) - 2$ multiplications.

(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 $i \in GF(p^k)^*$ as described above):
   - 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 $\{0,1\}$) 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

Thus, computing f_1 has a total effort of:

- **Field additions:** $2p^k - 2 + p^{2k} - 3p^k + 2 + p^k - 2 = p^{2k} - 2$
- **Field multiplications:** $2p^k \cdot \log_2(p^k) - 2\log_2(p^k) - 2 + p^k - 1$
  $= 2p^k \cdot \log_2(p^k) + p^k - 2\log_2(p^k) - 3$
- **Multiplicative depth:** $\lceil \log_2(p^k - 1) \rceil + 1$

And thus, combined with the effort for f_2, we get the costs of computing the carry r_i as

- **Field additions:** $p^{2k} + 2p^k - 3$
- **Field multiplications:** $2p^k \cdot \log_2(p^k) + 2p^k - 2\log_2(p^k) - 5$
- **Constant multiplications:** 1
- **Multiplicative depth:** $\max\{\lceil \log_2(p^k - 1) \rceil + 1, \mathbf{D}(r_{i-1})\} + 1$

C.1.2 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 $c_i = \tilde{f}(a_i, b_i, r_i) := a_i +_{\mathbb{Z}_{p^k}} b_i +_{\mathbb{Z}_{p^k}} r_i = (1 - r_i) \cdot \tilde{f}_1(a_{,i}, b_i) + r_i \cdot \tilde{f}_2(a_i, b_i)$ with

$$\tilde{f}_1(a,b) = a +_{\mathbb{Z}_{p^k}} b \quad \text{and} \quad \tilde{f}_2(a,b) = a +_{\mathbb{Z}_{p^k}} b +_{\mathbb{Z}_{p^k}} 1. \tag{23}$$

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 1). 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

$$g_1(a,b) = \tilde{f}_1(a,b) \quad \text{and} \quad g_2(a,b) = \tilde{f}_2(a,b) - \tilde{f}_1(a,b) \tag{24}$$

The actual degrees for each and the expected value can be found in Table 3, where the table for g_1 contains some additional entries that were used to check Formula 25. Missing entries are due to very long computation times in deriving the formulas (the code is available upon request).

We can see that the rule appears to be:

$$\deg(g_1(a,b)) = 2p^k - p^{k-1} - p^2 + p \tag{25}$$

and

$$\deg(g_2(a,b)) = p^k - p. \tag{26}$$

Since this is relatively close to the degree 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 24. Thus, we first compute the effort of $\tilde{f}_1$ and $\tilde{f}_1$, 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, keep track of the constant multiplications in step 3, and sum over all of $GF(p^k)$ in steps 3 and 4. Thus, 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 $l_j(b) \cdot \sum_{i \in GF(p^k)} l_i(a) \cdot \tilde{r}(i, j)$ 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: $p^k - 1$
   - Field multiplications: 0
   - Multiplicative depth: +0

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

- 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: $\lceil \log_2(p^k - 1) \rceil + 1$

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

C.1.3 Computing c_i. Putting the results from this section together, we get (with all operations in $GF(p^k)$):

$$\begin{aligned} c_i(a_i, b_i, r_{i-1}) &= g_1(a_i, b_i) + r_i \cdot g_2(a_i, b_i) \\ &= g_1(a_i, b_i) \\ &\quad + \left(f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1}) \right) \cdot g_2(a_i, b_i) \end{aligned} \tag{27}$$

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 mults: $\texttt{CMults}(g_1) + \texttt{CMults}(f_1) + \texttt{CMults}(f_2) + \texttt{CMults}(g_2)$

Inserting the values from the previous two subsections, we get an effort of

- **Field additions:** $p^{2k} + 2p^k - 3 + p^{2k} - 2 + 2p^k - 1 + p^{2k} + 2p^k - 2 + 2$
  $= 3 \cdot p^{2k} + 6p^k - 6$
- **Field multiplications:** $2p^k \cdot \log_2(p^k) + p^k - 4 + 2p^k \cdot \log_2(p^k) + p^k - 2\log_2(p^k) - 3 + p^k - 2 + 2p^k \cdot \log_2(p^k) + p^k - 4 + 2$
  $= 6p^k \cdot \log_2(p^k) + 4p^k - 2\log_2(p^k) - 11$
- **Constant multiplications:** $2p^{2k} + 1$

Regarding depth, we have
Depth =

$$\max\left\{\texttt{D}(g_1), \max\left\{\max\left\{\texttt{D}(f_1), \max\left\{\texttt{D}(r_{i-1}), \texttt{D}(f_2)\right\} + 1\right\}, \texttt{D}(g_2)\right\} + 1\right\}$$

from Equation 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 $r_i = \left(f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1})\right)$, as it is an input to the next round. Still, we use the samller value in our depth analysis, which can be found in Section C.2.

C.2 Adding Two 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 $\ell = \lfloor \log_p(x) \rfloor + 1$. 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: $3p^{2k} + 4p^k - 6$
  - Field multiplications: $6p^k \cdot \log_2(p^k) + 3p^k - 2\log_2(p^k) - 10$
  - Constant multiplications: $2p^{2k}$
  - Multiplicative depth: $\lceil \log_2(p^k - 1) \rceil + 2$
- $c_2 = g_1(a_2, b_2) + \left(f_1(a_1, b_1) + r_1 \cdot f_2(a_1, b_1)\right) \cdot g_2(a_2, b_2)$: 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 $\texttt{D}(c_2) = \lceil \log_2(p^k - 1) \rceil + 3$. Note that r_2 only has a degree of $\lceil \log_2(p^k - 1) \rceil + 2$, so this will make have no impact in reality.
- $c_i = g_1(a_i, b_i) + r_i \cdot g_2(a_i, b_i) \quad (2 < i < l)$
  $= g_1(a_i, b_i) + \left(f_1(a_{i-1}, b_{i-1}) + r_{i-1} \cdot f_2(a_{i-1}, b_{i-1})\right) \cdot g_2(a_i, b_i)$: 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

$\mathtt{D}(r_i) = \mathtt{D}(c_i)$ (at the cost of increased computation). Thus, we get:

- Field additions:$3 \cdot p^{2k} + 6p^k - 6$
- Field multiplications: $= 6p^k \cdot \log_2(p^k) + 4p^k - 2\log_2(p^k) - 11$
- Constant multiplications: $2p^{2k} + 1$
- Multiplicative depth: $\lceil \log_2(p^k - 1) \rceil + i$

- $c_\ell = r_\ell$:
  - Field additions: $p^{2k} + 2p^k - 3$
  - Field multiplications: $2p^k \cdot \log_2(p^k) + 2p^k - 2\log_2(p^k) - 5$
  - Constant multiplications: 1
  - Multiplicative depth: $\lceil \log_2(p^k - 1) \rceil + \ell$

To summarize (with `eff()` denoting above costs), we have the following effort:

Case 1: $0 \leq x \leq p^k - 1$**:** This means that our number can be encoded with 1 digit, and the result will have two digits, so the effort is $\mathtt{eff}(c_0) + \mathtt{eff}(r_1) = \mathtt{eff}(c_0) + \mathtt{eff}(f_1)$, so we have

- Field additions: $2p^{2k} + 2p^k - 5$
- Field multiplications: $4p^k \cdot \log_2(p^k) + 2p^k - 2\log_2(p^k) - 7$
- Constant multiplications: p^{2k}
- Multiplicative depth: $\lceil \log_2(p^k - 1) \rceil + 1$

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 $\mathtt{eff}(c_0) + \mathtt{eff}(c_1) + (\ell - 2) \cdot \mathtt{eff}(c_i) + \mathtt{eff}(c_\ell)$, so we get

- Field additions: $5p^{2k} + 8p^k - 12 + (\ell - 2) \cdot (3 \cdot p^{2k} + 6p^k - 6)$ $= (3\ell - 1) \cdot p^{2k} + (6\ell - 4) \cdot p^k - 6\ell$
- Field multiplications: $10p^k \cdot \log_2(p^k) + 6p^k - 4\log_2(p^k) - 19 + (\ell - 2) \cdot (6p^k \cdot \log_2(p^k) + 4p^k - 2\log_2(p^k) - 11)$ $= (6\ell - 2) \cdot p^k \cdot \log_2(p^k) + (4\ell - 2) \cdot p^k - 2\ell \cdot \log_2(p^k) - 11\ell + 3$
- Constant multiplications: $3p^{2k} + 1 + (\ell - 2) \cdot (2p^{2k} + 1)$ $= (2\ell - 1) \cdot p^{2k} + \ell - 1$
- Multiplicative depth: $\lceil \log_2(p^k - 1) \rceil + \ell$

As presented in Section 6 and illustrated by Figure 4, we conclude that the p^k-encoding performs very poorly regarding all metrics, and using $\mathbb{Z}_p$ as an encoding base is the better choice.