Accelerating Homomorphic Computations on Rational Numbers

Angela Jäschke[1] and Frederik Armknecht[1]

[1]University of Mannheim
{jaeschke, armknecht}@uni-mannheim.de

Abstract

Fully Homomorphic Encryption (FHE) schemes are conceptually very powerful tools for outsourcing computations on confidential data. However, experience shows that FHE-based solutions are not sufficiently efficient for practical applications yet. Hence, there is a huge interest in improving the performance of applying FHE to concrete use cases. What has been mainly overlooked so far is that not only the FHE schemes themselves contribute to the slowdown, but also the choice of data encoding. While FHE schemes usually allow for homomorphic executions of algebraic operations over finite fields (often $\mathbb{Z}_2$), many applications call for different algebraic structures like signed rational numbers. Thus, before an FHE scheme can be used at all, the data needs to be mapped into the structure supported by the FHE scheme.

We show that the choice of the encoding can already incur a significant slowdown of the overall process, which is independent of the efficiency of the employed FHE scheme. We compare different methods for representing signed rational numbers and investigate their impact on the effort needed for processing encrypted values. In addition to forming a new encoding technique which is superior under some circumstances, we also present further techniques to speed up computations on encrypted data under certain conditions, each of independent interest. We confirm our results by experiments.

This work is the extended version of ([23], DOI 10.1007/978-3-319-39555-5_22, Springer copyright). It differs in several key aspects: Operations are discussed in much greater depth and explained through examples, we show how to derive the theoretical formulas we give, further concrete numbers are presented, and we explicitly show how to transform a function into a boolean circuit. We also extended the background on Machine Learning to make this paper more easily accessible and show how to reduce redundancy in multiplication through concrete formulas.

1 Introduction

Fully Homomorphic Encryption (FHE) is a very promising field of research because it allows arbitrary computations on encrypted data. This means that data can be outsourced securely (e.g. into the cloud) without sacrificing functionality, as any operation one would like to perform on the data can also be performed on the encrypted data by a third party without divulging information. With a powerful enough encryption scheme, this third party may even apply its own proprietary algorithm, like a machine learning algorithm, to the encrypted data such that the result divulges nothing about the algorithm that was applied except for what can be inferred from the result itself - this is the setting we will assume. While multiparty computation also offers this kind of confidential computation, it requires frequent interaction between the involved parties, which seems unfortunate for the goal of outsourcing computation. For this reason, we instead focus on FHE, which allows a non-interactive solution. Unfortunately, FHE-based solutions today are still very slow and thus not very practical. Current FHE schemes are all based on Somewhat Homomorphic Encryption (i.e., allow a limited number of operations) and are noise-based. Every multiplication increases the amount of noise, and if the noise exceeds a certain threshold, the ciphertexts become undecryptable. There are two approaches to accommodate arbitrary many ciphertext multiplications in these schemes: In so-called leveled FHE schemes, one can adjust the encryption scheme to support a predetermined number of consecutive multiplications (multiplicative depth), where the scheme becomes slower the larger the depth is. For this reason, minimizing depth is one of our goals in this paper. The other approach for handling the problems that come with consecutive multiplications, which we opted for because of very large depths in our use cases, is called bootstrapping. Here, the ciphertext is "cleaned up" after multiplication, but this operation takes very long and constitutes the main bottleneck when used. For this reason, minimizing the total number of multiplications is another of our goals.

Because of these efficiency problems, there is currently much research on improving the efficiency of the schemes themselves on the one hand, and on designing algorithms that are particularly suited to FHE, i.e., through minimal multiplicative depth, on the other hand. While this is certainly a valuable contribution for some use cases, we feel that in general the algorithms one wants to perform on the data are predetermined and not up for discussion. At first glance, this might seem to imply that there is little potential for improvement apart from improving the schemes themselves, but we show that this is indeed not the case.

Generally, suppose one has an FHE scheme $\mathcal{E} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with plaintext space $\mathcal{M}$ and ciphertext space $\mathcal{C}$, and there is a function $g : \mathcal{M}^z \to \mathcal{M}$ for some $z \in \mathbb{N}$. Then a Fully Homomorphic Encryption scheme promises that there exists a corresponding function $g^* : \mathcal{C}^z \to \mathcal{C}$ with

$$\mathsf{Dec}(\mathsf{sk}, g^*(\mathsf{Enc}(\mathsf{pk}, m_1), \ldots, \mathsf{Enc}(\mathsf{pk}, m_z))) = g(m_1, \ldots, m_z).$$

However, plaintext spaces for encryption schemes are usually some finite field $GF(p^d)$ for some prime p and power d, so if we want to work with elements from a different structure S (like the rational numbers), we must first map them[1] to the plaintext space using an encoding $\pi : S \to \mathcal{M}^k$ and then perform a function on the plaintext values that emulates the function on S. For a better understanding, suppose we have an encryption scheme like above. Then, if we want to evaluate a function $f : S^n \to S$ on encrypted data, we must first turn f into a function $g : (\mathcal{M}^k)^n \to \mathcal{M}^k$ on the plaintext space (where $\mathcal{M}^k$ emulates S) and then execute the function $g^* : (\mathcal{C}^k)^n \to \mathcal{C}^k$ that corresponds to g. This is illustrated in Figure 1.

$$\begin{tikzcd}
\text{Function Space:} & S^n \arrow[r, "f"] \arrow[d, "\pi"] & S \\
\text{Plaintext Space:} & (\mathcal{M}^k)^n \arrow[r, "g"] \arrow[d, "\mathsf{Enc}"] & \mathcal{M}^k \arrow[u, "\pi^{-1}"'] \\
\text{Ciphertext Space:} & (\mathcal{C}^k)^n \arrow[r, "g^*"] & \mathcal{C}^k \arrow[u, "\mathsf{Dec}"']
\end{tikzcd}$$

Figure 1: Steps in homomorphic evaluation

As it turns out, there is often no unique function g for a given function f, but instead several different ones which depend on the chosen encoding function π. This also means that the most we can aim for in terms of efficiency in evaluating a function f on encrypted data is not f itself, but rather its emulation g on the plaintext space. As it turns out, the increase here is not negligible: While the Perceptron, which we evaluate in Section 6.3 on encrypted data, runs almost instantaneously (roughly 0.004 seconds) for ten rounds when computing on unencrypted rational numbers, the evaluation of the same algorithm emulated on the plaintext space (i.e., still unencrypted) takes over 120 seconds for the same parameters even with our most efficient encoding in the plaintext space. This shows that though largely ignored until now, the overhead that comes from switching from the function f to g can be substantial and must equally be addressed to make FHE applications as efficient as possible. Thus, while previous work on making computations with FHE more efficient has focused primarily on the area inside the dashed red rectangle in Figure 1, we investigate how to improve efficiency through the right choice of π and subsequently g, represented by the solid green rectangle. Motivated by the idea of outsourcing actual data

[1]For example, if $S = \{x \in \mathbb{Z} | 0 \leq x \leq 7\}$ (i.e., numbers representable by 3 bits) but the plaintext space of the encryption scheme is only $\mathcal{M} = \{0, 1\}$, we could map $\pi : S \to \mathcal{M}^3$.

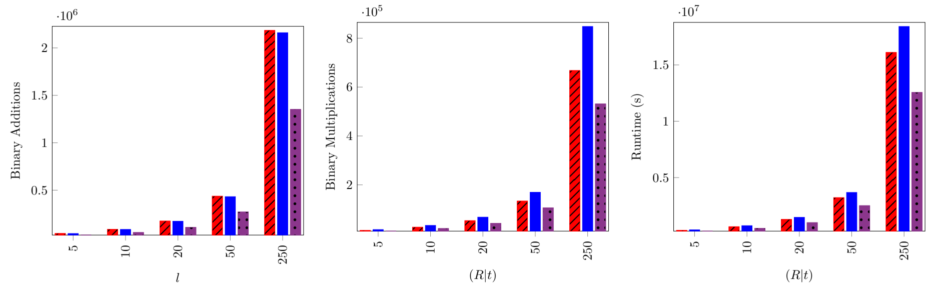
and running existing algorithms on it, we face the challenges of encoding rational numbers (as opposed to elements of finite fields or unsigned integers) and of incorporating basic operations like addition, multiplication and comparison, which are needed for many popular algorithms.

1.1 Our Contribution

In this paper, we address the above challenges and try to minimize total number of multiplications (and the multiplicative depth) of g through appropriate choices in π. We also examine some further optimizations which increase efficiency under certain assumptions and are of independent interest. As a concrete application, we apply our results to two use cases from machine learning, the Perceptron and the Linear Means Classifier, and see that the right choice of π can make a significant difference in terms of multiplicative depth, total number of multiplications, and in terms of runtime, for which we encrypted the data with the HElib library. To this end:

- We present a new method for working with encrypted rational numbers by solving the problem that the number of digits of precision doubles with each multiplication. We show how to remove the extra digits and bring the number back down to a predefined precision level, greatly improving performance without leaking information about the function that was applied.
- We investigate two different popular encodings with regard to efficiency in emulating basic operations on rational numbers like comparison, addition and multiplication, and present a hybrid encoding that surpasses the two traditional ones both in theory (as measured by total bit additions, multiplications and required multiplicative depth) and in terms of actual runtime for large sizes.
- We show how to derive the boolean circuit (i.e., the polynomial over $GF(2)$) for comparing two encrypted numbers, and present an easier way for comparing numbers to 0 which takes almost no time.
- We show how to increase efficiency in the case that the numbers are bounded, like in real-world applications where values lie in some known range.
- We confirm our results by implementing the Perceptron, an important fundamental algorithm in machine learning, and running it using the different encodings, as well as a polynomial like that used for Linear Means Classification.

As a quick preview, consider Figure 2, which shows theoretical bounds on the number of bitwise additions and multiplications as well as extrapolated runtime needed to apply a Linear Means Classifier with each of the three encodings for different numbers of features. We can see that our new hybrid encoding mechanism is superior in all three aspects, making it an attractive choice.



(a) Bit Additions. (b) Bit Multiplications. (c) Runtime (s)

Figure 2: Bounds for the number of bitwise additions and multiplications as well as runtime for evaluating Linear Means Classifier with l features of length 30 for different l using Two's Complement ● (lines), Sign-Magnitude ● (solid) and Hybrid Encoding ● (dotted)

1.2 Outline

We start by giving an overview of related work in Section 2. In Section 3, we give some background on Fully Homomorphic Encryption and the challenges faced when working with rational numbers, as well as on the two encodings we use. In Section 4, we show how to emulate the addition, multiplication and comparison of encoded numbers using just binary additions and multiplications and analyze complexity. Section 5 presents different ways of accelerating computations on encrypted data, and Section 6 gives some motivation and necessary background on machine learning before using two algorithms from this field to demonstrate the effects of our improvements. Lastly, Section 7 gives our conclusion, and an insight into future work. The appendix shows how to turn functions into Boolean circuits.

2 Related Work

While encryption schemes that allow one type of operation on ciphertexts are well understood and have a comprehensive security characterization ([4]), Fully Homomorphic Encryption, which allows both unlimited additions and multiplications, was only first solved in [18]. Since then, numerous other schemes have been developed, for example [28], [10], [9], [26], [13], [14] and [20]. An overview can be found in [3]. There have been several works concerning actual implementation of FHE, like [19] (homomorphically evaluating the AES circuit), [7] (predictive analysis on encrypted medical data), or [21] (machine learning on encrypted data), and there are two publicly available libraries ([1],[17]). [24] discusses whether FHE will ever be practical and gives a number of possible applications, including encrypted machine learning. Most recently, two publications regarding encoding rational numbers for FHE have appeared, illustrating what an important topic this is: [12] examines encoding rational numbers through continued fractions (restricted to positive rationals and evaluating linear multivariate polynomials), whereas [15] focuses on most efficiently embedding the computation into a single large plaintext space. Another work that explores similar ideas as [15] and also offers an implementation is [16].

While the idea of being able to privately evaluate machine learning algorithms is certainly intriguing, the overwhelming majority of work in this area considers multiparty computation, which requires interaction between the client and the server during computation and is thus a different model. Examples include [29], [25] and [8], and works like [22] and [27] concern themselves with efficiency measures and circuit optimizations specific to multiparty computation. Another line of research regarding confidential machine learning, e.g. [7] and again [8], focuses on a scenario where the model being computed and/or evaluated is publicly known - a scenario we explicitly exclude. Other work like [11] restricts itself to unsigned integers, making all involved circuits much less complex. Work like [5] considers recommender systems, but in a scenario which becomes insecure if too many fresh encryptions are available. Closest to our work is [21], which restricts itself to machine learning algorithms like the Linear Means Classifier and Fishers Linear Discriminant Classifier, which can be expressed as polynomials of low degree, and focuses on the classification, not the derivation of the model. Their encoding of input data is also restricted to functions with few multiplications.

We stress that until now, all approaches dealing with rational numbers do not fully solve the problem, as computations are either restricted to positive integers, or the multiplicative depth of the computation must be know beforehand. Our approach is the first to actually tackle the problem of computing on rational numbers with no further assumptions, and offers other improvements if some assumptions can be made.

3 Background

3.1 FHE and Efficiency Metrics

Fully Homomorphic Encryption (FHE) describes a class of encryption schemes that allow arbitrary operations on encrypted data. This would, in theory, enable outsourcing of encrypted data to an untrusted cloud service provider, who could still perform any operations the user wishes. This means that we can protect privacy (as opposed to uploading the data in unencrypted form) while maintaining functionality (as opposed to uploading data encrypted under conventional schemes). Unfortunately, FHE today it is still rather slow, although huge advancements have been made in the last six years.

Because of this, one of our measures for efficiency is the number of bit additions and multiplications performed, as this would translate directly into the number of homomorphic additions and multiplications performed if the data were encrypted. Note that in schemes today, homomorphic multiplication tends

to be computationally more expensive than addition.
In our analysis of computational effort, we also include the multiplicative depth: Many publications today use *Leveled Fully Homomorphic Encryption*, which is related to Fully Homomorphic Encryption in that arbitrary functions f can be performed on the encrypted data, but the multiplicative depth of f must be known beforehand, and efficiency of the encryption scheme decreases as this number increases. Multiplicative depth measures how many consecutive multiplications are performed. For example, the polynomial $x_1 \cdot x_2 + x_1 \cdot x_3 + x_2 \cdot x_3$ has 3 multiplications in total, but a multiplicative depth of only 1. These leveled schemes can be more efficient than pure FHE schemes for small depths, but if more than the allowed number of consecutive multiplications are performed, decryption may return the wrong result. To this end, we include multiplicative depth in our analysis and aim to minimize it as one of our goals. We would, however, like to point out that if one uses bootstrapping, as we did in our implementations, depth becomes less of an issue and the total number of multiplications is the main factor determining runtime.

3.2 From Unsigned Integers to Rationals of Arbitrary Precision

In previous work (e.g. [6], see also Section 2), rational numbers have often been approximated by multiplying with a power of 10 and rounding, but note that when multiplying two rational numbers with k bits of precision, we obtain a number with $2k$ bits of precision (whereas addition does not change the precision). If we are working on unencrypted numbers, we might just round to obtain k bits of precision again, or we could truncate (truncation after k bits yields the same accuracy as rounding to $k-1$ bits). However, things become more difficult if we will be operating on encrypted data, as rounding is generally not possible here and thus these extra bits of precision accumulate. To see this, suppose a precision of k digits is required. One would usually multiply the rational number with 10^k and round (or truncate) to the nearest integer, which is then encoded and encrypted. Dividing the decrypted decoded number by 10^k again yields the rounded rational. However, the problem of doubling precision with multiplication is prevalent here. Consider what would happen if we were to multiply two such numbers: Suppose we have two rational numbers a and b that we would like to encode as integers a' and b' with k digits of precision, so we get $a' = a \cdot 10^k$ and $b' = b \cdot 10^k$ (rounded to the nearest integer). Multiplying a' and b', we get $c'' = a' \cdot b' = a \cdot 10^k \cdot b \cdot 10^k = (a \cdot b) \cdot 10^{2k}$. Thus, having reversed the encoding, the obtained value c'' must be divided by 10^{2k}. This is a problem because we cannot remove the extra bits by dividing by 10^k, so the party performing the algorithm must now divulge what power of 10 to divide the obtained result by. This leaks information about the multiplicative depth of the function used and thus constitues a privacy breach for the computing party. Additionally, there is also the problem during computation that the sizes of the encoded numbers will increase substantially.

To solve this problem, we propose the following approach: Instead of scaling by a power of 10, we multiply by a power of 2 and truncate to obtain an integer that we will encode in binary fashion, so that we can later encrypt each bit separately. This eliminates the above problem: Multiplying two numbers a' and b' with k bits of precision still yields $c'' = (a \cdot b) \cdot 2^{2k}$, but since we are encoding bit by bit, dividing by 2^k and truncating corresponds to merely deleting the last k (encrypted) bits of the product. Thus, the party performing the computations can bring the product c'' back down to the required precision after every step by discarding the last k bits and thus obtaining $c' = a \cdot b \cdot 2^k$, meaning that the party which holds the data must always divide the decoded result by 2^k no matter what operations were applied. This has the benefit of not only hiding the data from the computing party, but also hiding the function from the party with the data.

3.3 Two's Complement

Having determined that we will be encoding bit for bit to support arbitrary precision without information leakage, we must now decide on how exactly we want to represent a rational number (which has been scaled to be a signed integer). For unsigned integers, binary representation is well known: Given an integer $a \geq 0$, we write it as $a = \sum_{i=0}^{n} a_i \cdot 2^i$ where $n = \lfloor \log_2(|a|) \rfloor$ and $a_i \in \{0, 1\}$ to obtain a $n+1$-bit string $a_n a_{n-1} \ldots a_1 a_0$.

To incorporate negative numbers, the most popular encoding is called *Two's Complement*: Here, we write an integer a as $a = a_{n+1} \cdot (-2^{n+1}) + \sum_{i=0}^{n} a_i \cdot 2^i$ where $n = \lfloor \log_2(a) \rfloor$ and $a_i \in \{0, 1\}$. This means that the most significant bit (MSB) encodes the negative value -2^{n+1} and is thus 1 exactly when $a < 0$. As an example, consider the bitstring 1011, which encodes $1 \cdot (-2^3) + 0 \cdot 2^2 + 1 \cdot 2 + 1 \cdot 1 = -8 + 2 + 1 = -5$. The most important operations for numbers encoded in this fashion are presented in the following.

3.3.1 Addition

Addition works in much the same way as normal binary addition, except for one point: To obtain the correct result when adding two n-bit numbers, the result must also be encodeable by n bits, and any values past the n^{th} bit are discarded. Since the result of adding two n-bit numbers is usually $n+1$ bits long, we first extend the inputs by one bit without changing their value so that we can then add two $n+1$ bit numbers whose sum is also encodeable by $n+1$ bits, thus yielding the correct result. Note that to extend the bitlength by k, we merely replace the most significant bit by $k+1$ copies of itself (so 1 for a negative number and 0 otherwise). This is called *sign extension.* As an example, suppose we are adding the 4-bit numbers $-5 = 1011$ and $7 = 0111$: First, we use sign extension to obtain the 5-bit numbers $-5 = 11011$ and $7 = 00111$, then we carry out normal addition, resulting in the number 100010. Discarding the excess leftmost bit, we obtain 00010, which indeed decodes to $2 = -5 + 7$.

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

3.3.2 Multiplication

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

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

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

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

As an example, we will multiply $-3 = 101$ ($m = 3$) and $7 = 0111$ ($n = 4$):

$$\begin{array}{cccccccc|ccccccc}
1 & 1 & 1 & 1 & 1 & 0 & 1 & \cdot & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
\hline
 & & & & & & & & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
 & & & & & & & & 1 & 1 & 1 & 1 & 0 & 1 & \\
 & & & & & & & & 1 & 1 & 1 & 0 & 1 & & \\
 & & & & & & & & 0 & 0 & 0 & 0 & & & \\
 & & & & & & & & 0 & 0 & 0 & & & & \\
 & & & & & & & & 0 & 0 & & & & & \\
 & & & & & & & & 0 & & & & & & \\
\hline\hline
 & & & & & & & & 1 & 1 & 0 & 1 & 0 & 1 & 1
\end{array}$$

This correctly yields $1101011 = -21$. Whenever we refer to the addition of rows during multiplication in this paper, we are referring to the rows as in the above example, which has 7 rows of lengths $7, 6, \ldots, 1$, respectively.

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

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

Here, we can see that out of the 10 terms that occur (generally: $(n+m)+(n+m-1)+\cdots+2+1=\frac{(n+m)\cdot(n+m+1)}{2}$), the term $a_0 \cdot b_0$ occurs once, and the terms $a_1 \cdot b_0$, $a_0 \cdot b_1$ and $a_1 \cdot b_1$ all occur three times. As can easily be seen, we actually only have $n \cdot m$ different products, so we can save a significant amount of computation by avoiding this redundancy. A detailed explanation of how to do this can be found in Section 5.3.

3.3.3 Negation

Negating a given number is simple in Two's Complement:

i. Flip all bits (i.e., XOR them with 1).

ii. Add 1 to the resulting number.

As an example, take the number $-3 = 101$: Flipping all bits gives us 010, and adding 1 yields $011 = 3$.

3.4 Sign-Magnitude

While Two's Complement may be the most popular encoding of signed integers, it is not the only one: *Sign-Magnitude* encoding formalizes the most intuitive idea of having an extra bit that determines the sign. Conventionally, this is the most significant bit, which is 1 when a number is negative and 0 when a number is positive. Thus, for example, the number $5 = 0101$ and $-5 = 1101$. This notation suffers from the fact that there are two encodings of 0 ($0 = 00\dots00$ and $-0 = 10\dots00$) and is seldom used, but we will later see how this slightly unconventional encoding can help us. A detailed presentation of the individual operations can be found below.

3.4.1 Addition

Addition is surprisingly complex for Sign-Magnitude and we will need several subroutines:

- `NormalAdd`(a, b): a and b are positive integers in regular binary notation (which is the same as Sign-Magnitude for positive numbers) and regular binary addition is performed.

- `NormalSub`(a, b): a and b are positive integers with $a \geq b$ in regular binary notation and regular binary subtraction $a - b$ is performed (using the Complement Method).

- `SMCompare`(a, b): a and b are arbitrary integers, the function returns 1 exactly when $a \leq b$. Further details on this function can be found in Appendix A.2.

- `SMNeg`(a): Return $-a$ (see below).

With these subroutines, we can express addition of a and b, both of length $n+1$, as in Figure 3. Note that a_n and b_n denote the sign, $\tilde{a} = a_{n-1} \dots a_1 a_0$ (i.e., with discarded sign bit) and $a_n || c$ means appending a's sign to a string c.

```
1: SMBinAdd()
  Input: $a = a_n a_{n-1} \dots a_1 a_0$ and $b = b_n b_{n-1} \dots b_1 b_0$
2:     if $a_n = b_n$ then
3:         $c = a_n || \texttt{NormalAdd}(\tilde{a}, \tilde{b})$
4:         (Add absolute values and append sign)
5:     end if
6:     if $a_n = 1$ and $b_n = 0$ then
7:         $\texttt{SMCompare}(\texttt{SMNeg}(a), b)$
8:         (compare absolute values)
9:         if $\texttt{SMNeg}(a) \leq b$ then
10:            $c = b_n || \texttt{NormalSub}(b, \texttt{SMNeg}(a))$
11:        else
12:            $c = a_n || \texttt{NormalSub}(\texttt{SMNeg}(a), b)$
13:        end if
14:        (subtract smaller value from larger value and append sign of larger one)
15:    end if
16:    if $a_n = 0$ and $b_n = 1$ then
17:        $\texttt{SMCompare}(a, \texttt{SMNeg}(b))$
18:        (compare absolute values)
19:        if $a \leq \texttt{SMNeg}(b)$ then
20:            $c = b_n || \texttt{NormalSub}(\texttt{SMNeg}(b), a)$
21:        else
22:            $c = a_n || \texttt{NormalSub}(a, \texttt{SMNeg}(b))$
23:        end if
24:        (subtract smaller value from larger value and append sign of larger one)
25:    end if
  Output: $c$
26: end
```

Figure 3: Addition in Sign-Magnitude encoding

3.4.2 Multiplication

Multiplication with Sign-Magnitude encoding is conceptually very simple: We delete the sign bits a_n and b_n, multiply the remaining bitstrings as with regular binary multiplication, and append the sign bit $a_n \oplus b_n$. Note that to add the individual rows that we obtain, we do not have to use the complicated addition of section 3.4.1 but rather the much more efficient subroutine `NormalAdd()`.

3.4.3 Negation

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

4 Basic Operations and Their Performance

Having introduced two different ways of encoding, this section will now examine both the theoretical complexity and actual performance of elementary operations. All computations were done on a virtual machine with 5 GB of RAM running Ubuntu 14.04 LTS (running on a Lenovo Yoga 2 Pro with a Intel i7-4500U processor with 1.8 GHz and 8 GB of RAM with Windows 8.1). We give the number of binary additions and multiplications as well as multiplicative depth required for these elementary operations. Due to space limitations, we omit how these values were determined, but two examples can be found in Appendix A along with a detailed explanation on how to turn a function into a boolean circuit. We note that we also implemented all our functions with a subroutine that counts these values to ensure that the formulas are correct. Runtimes were obtained for values encrypted with the HElib library ([1]).

4.1 Note on Comparisons

Since Sign-Magnitude uses a comparison in its addition function, Appendix A.2 shows how to implement this comparison as a polynomial. We note, however, that when comparing a number with 0, there is an easier way (see Section 5.2). For the general case, the effort of comparing two arbitrary numbers is:

Two's Complement:

- $3n$ binary additions
- $n+1$ binary multiplications
- a multiplicative depth of n

Sign-Magnitude:

- $10n-3$ binary additions
- $6n-2$ binary multiplications
- a multiplicative depth of $2n-1$

We can see that Two's Complement is more efficient for comparing encrypted numbers.

4.2 Addition

We will now compare addition of two n-bit numbers for Two's Complement and Sign-Magnitude encoding. The computational effort is:

Two's Complement:

- $5n-2$ binary additions
- n binary multiplications
- a multiplicative depth of n

Sign-Magnitude:

- $73n-17$ binary additions
- $28n+4$ binary multiplications
- a multiplicative depth of $2n+2$

As we can see, Two's Complement again does better in theory. In practice (i.e., counted by our program), we get as values the number of operations and runtime as shown in Figure 4. These diagrams show that Two's Complement is indeed superior to Sign-Magnitude where addition is concerned.

4.3 Multiplication

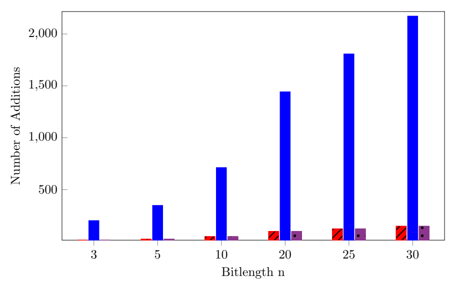
In this section, we will examine the multiplication of an n-bit number with an m-bit number. Heuristically, we expect Sign-Magnitude to do better here: Instead of the costly "normal" Sign-Magnitude addition operation which uses a comparison circuit, we can use regular textbook binary addition to add up the rows encountered in multiplication, so the fact that addition of two n-bit Sign-Magnitude numbers is much more expensive than that of two n-bit Two's Complement numbers does not weigh in here. On the other hand, because of the sign extension necessary in Two's Complement multiplication, not only are the rows longer ($n + m$ as compared to n), but there are also more of them ($n + m$ as opposed to m), so we must do more additions of longer bitstrings. We examine the effort required:
Two's Complement:

- $\frac{5(m^2+n^2)-19(m+n)}{2} + 5mn + 10$ binary additions
- $\frac{(m+n-3)(m+n)}{2} + mn + 1$ binary multiplications
- a multiplicative depth of $\lceil \log_2(m+n) \rceil \cdot (m+n-1) - 2^{\lceil \log_2(m+n) \rceil} + 2$

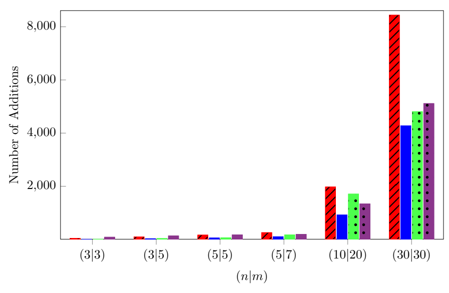
Sign-Magnitude:
Due to changing intermediate lengths during row additions (which depend on both n and m instead of just $n+m$ as in Two's Complement), an exact formula would be very involved and hardly informative. Thus, we present a formula for an upper bound which already shows that SM is superior to TC for multiplication. To this end, we now have two data sets for Sign-Magnitude in the diagrams 4b, 4d and 4f in Figure 4 regarding the number of operations: One shows the exact numbers as counted by an instruction in our program (and verified manually), and one shows the bounds as given by the following formulas:

- $(2^{\lceil \log_2(m-1) \rceil} - 1) \cdot (5n - 7) + (2^{\lceil \log_2(m-1) \rceil - 1} - 1) \cdot 5 \cdot \lceil \log_2(m-1) \rceil$ binary additions at most
- $(n-1) \cdot (m-1) + (2^{\lceil \log_2(m-1) \rceil} - 1) \cdot (n-1) + (2^{\lceil \log_2(m-1) \rceil - 1} - 1) \cdot \lceil \log_2(m-1) \rceil$ binary multiplications at most
- A multiplicative depth of at most
$\frac{1}{2} \lceil \log_2(m-1) \rceil \cdot (\lceil \log_2(m-1) \rceil + 2n - 5) + 2^{\lceil \log_2(m-1) \rceil}$

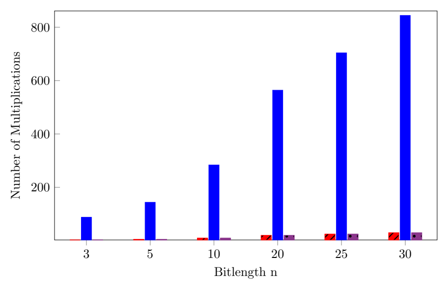
Concrete values and runtimes can be seen in Figure 4 and as we can see, Two's Complement performs much worse, as expected. Thus, Two's Complement encoding is superior for addition and comparison, but inferior for multiplication.



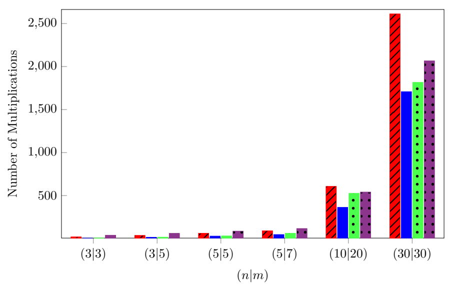
(a) Bit Additions (+).



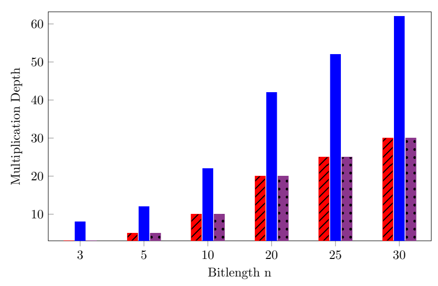
(b) Bit Additions (*).



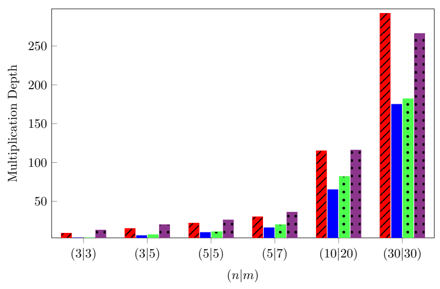
(c) Bit Multiplications (+).



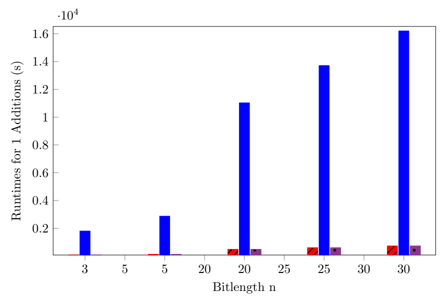
(d) Bit Multiplications (*).



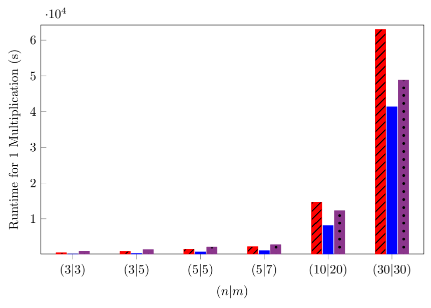
(e) Multiplicative Depth (+).



(f) Multiplicative Depth (*).



(g) Runtime (+).



(h) Runtime (*).

Figure 4: Comparison of addition (+) and multiplication (*) for Two's Complement ● (lines), exact values for Sign-Magnitude (counted by program) ● (solid), upper bound for Sign-Magnitude for multiplication● (dotted) and our new Hybrid Encoding (● (dotted). Runtimes for data encrypted with HElib.

5 Accelerating Computations

In this section, we will discuss several optimizations to make computations on encrypted data more efficient.

5.1 Hybrid Encoding

Since we have seen in the previous sections that Two's Complement encoding always performs better than Sign-Magnitude except for multiplication (where it is much worse), we propose the following approach, called Hybrid Encoding: We work with Two's Complement encoding, but when we want to multiply, we convert the numbers to their representations in Sign-Magnitude, perform the multiplication there, and convert the result back. As we will see, this is indeed more efficient than regular Two's Complement multiplication. To do this, we must first determine how to convert numbers from their representation in Two's Complement to their Sign-Magnitude form and vice versa, so suppose we have a number a under one encoding α (either Two's Complement or Sign-Magnitude), denoted a_α, and wish to transform it into its representation under the other encoding β, denoted a_β. For numbers with MSB 0, both encodings are actually the same ($a_\alpha = a_\beta$), so in this case we do nothing. If the number has a MSB of 1, we compute its negation ($a_\alpha \mapsto -a_\alpha$), which is the same for both encodings as it has MSB 0 ($-a_\alpha = -a_\beta$). We then negate the negation under the new encoding ($-a_\beta \mapsto a_\beta$), obtaining the original value in the new encoding.

As can easily be seen, the overhead we incur in addition to the cost of a Sign-Magnitude multiplication for multiplying two numbers of lengths n and m is basically that of 3 Two's Complement inversions, 3 Sign-Magnitude inversions (both of lengths n, m and $n+m$), and the cost of multiplying the boolean values representing whether the different cases are true or false. In total, the overhead costs (i.e., those incurred in addition to the costs for the Sign-Magnitude multiplication) are:

- $14(n+m) - 7$ binary additions
- $6(n+m) - 3$ binary multiplications
- a multiplicative depth of $\max\{n, m\} + 1 + n + m$

We present some concrete values for this overhead and runtimes in Figure 4 along with the same values for Two's Complement multiplication and Sign-Magnitude multiplication. The exact numbers can also be found in Tables 1 and 2.

As can easily be seen, HE performs better than Two's Complement in all aspects for multiplying large numbers, but is (naturally) not quite as good as Sign-Magnitude. The runtimes are roughly as we would expect from these numbers, i.e., the new multiplication is faster than Two's Complement for large numbers, but naturally slower than Sign-Magnitude.

n	m	TC+	TC*	TC*d	SM+	SM*	SM*d	O+	O*	O*d	HE+	HE*	HE*d
3	3	43	19	9	8	6	3	77	33	10	85	39	13
3	5	94	36	15	29	15	6	105	45	14	134	60	20
5	5	165	61	22	59	29	10	133	57	16	192	86	26
5	7	256	90	30	105	47	16	161	69	20	266	116	36
10	20	1975	606	115	924	363	65	413	177	51	1337	540	116
30	30	8440	2611	292	4279	1708	175	833	357	91	5112	2065	266

Table 1: Number of binary additions (TC+ and SM+), binary multiplications (TC* and SM*) and multiplicative depth (TC*d and SM*d) for multiplication of a n-bit number with an m-bit number in Two's Complement (TC) and Sign-Magnitude (SM) encoding, as well as additional cost to the SM-values we expect when using Hybrid Encoding multiplication (O+, O* and O*d), and total costs for Hybrid Encoding (HE+, HE* and HE*d).

n	m	TC	SM	HE
3	3	458	144	926
3	5	870	288	1359
5	5	1476	719	2076
5	7	2177	1060	2703
10	20	14731	8078	12224
30	30	62982	41362	48791

Table 2: Runtimes in seconds for 1 execution of the three different multiplications (encrypted).

Thus, we have found a new way to improve efficiency for large bitlengths: do all operations in Two's Complement notation, but switch to Sign-Magnitude for multiplication. We shall see the benefits of this in our real-world application in Section 6.3, though we would like to note that there may be applications where Sign-Magnitude is favorable (when there are very few additions). However, since in Fully Homomorphic Encryption, multiplicative depth is often key (as mentioned in Section 3.1) and bootstrapping is the bottleneck, our new approach seems favorable for large parameters under this aspect as well.

5.2 Easy Comparison

Apart from numerical computations, many algorithms require a comparison of two numbers, which would usually require a rather expensive computation. However, we argue that in some use cases where one only has to compare a number to 0, like in the Perceptron, there is a much easier way. Instead of computing a costly circuit for comparison, it suffices to take the most significant bit of the number, which will be 0 if the number is greater than zero and 1 if it is less. For Two's Complement, it will be 0 also when the number equals 0, but in Sign-Magnitude it can be either 0 or 1 when using this method, as there are two encodings of 0 here. Thus, if the sum is exactly 0, the resulting bit is wrong for Two's Complement and can be either case for Sign-Magnitude. We observe, however, that when initializing the weights $w_1, \dots, w_l$ with random rational numbers, a weighted sum $w_1x_1 + \cdots + w_lx_l$ is highly unlikely to be 0. Thus, in this case there should be no change whether the condition for an operation is $w_1x_1 + \cdots + w_lx_l > 0$ or $w_1x_1 + \cdots + w_lx_l \geq 0$ and the easy comparison should return the correct result with overwhelming probability. If the weights are initialized with 0 (as could be chosen in the Perceptron) or integers in the more general case, a more involved formula like that in Appendix A.2 should be used.

5.3 Improved Multiplication

As already mentioned, the sign extension in Two's Complement introduces costly redundancy. This section now shows in detail how to avoid computing the same product several times, which more than halves the computation effort of the matrix computation step by bringing it from $\frac{(n+m)\cdot(n+m+1)}{2}$ to $n \cdot m$. To this end, we will think of the rows that are generated as an $(n+m) \times (n+m)$ - matrix A (with some empty entries) indexed as $A[i,j]$, where i refers to the row and j to the bit position in the row, i.e., 0 is on the right-hand side. As an example for the case with $n = m = 2$ (where we write $[i,j]$ instead of $A[i,j]$):

$$\begin{array}{ccccccccc} a_1 & a_1 & a_1 & a_0 & \cdot & b_1 & b_1 & b_1 & b_0 \\ \hline & & & & & [0,3] & [0,2] & [0,1] & [0,0] \\ & & & & & [1,3] & [1,2] & [1,1] & \\ & & & & & [2,3] & [2,2] & & \\ & & & & & [3,3] & & & \end{array}$$

With this notation in place, we present the following pseudocode instructions (Figure 5) for reducing the number of computations in multiplying two numbers of lengths n and m. Note that the below code eliminates all superfluous product computations when $m = n$. When $m < n$, we have $b_j = b_{m-1}$ also for $m \leq j \leq n - 1$. Though not explicitly stated below for reasons of readability, it is not hard to see how the code can be extended to eliminate this redundancy as well.

```
 **Input:** $a_{n-1} \dots a_1 a_0,\ b_{m-1} \dots b_1 b_0$

for $0 \leq i \leq n-2$ do
    for $0 \leq j \leq n-1$ do
        $A[i, j+i] = b_i \cdot a_j$
    end for
    for $n \leq j \leq n+m-1-i$ do
        $A[i, j+i] = A[i, n+i-1]$
    end for
end for
for $0 \leq j \leq n-2$ do
    $A[n-1, n+j-1] = b_{n-1} \cdot a_j$
    for $n \leq i \leq n+m-j-1$ do
        $A[i, i+j] = A[n-1, n+j-1]$
    end for
end for
$A[n, n+m-1] = b_{n-1} \cdot a_{n-1}$
$A[n-1, n+m-1] = A[n, n+m-1]$
$A[n-1, n+m-2] = A[n, n+m-1]$
 Output: $A$
```

Figure 5: Redundancy in Multiplication

Of course, as Sign-Magnitude multiplication works without sign extension, this improvement only applies to Two's Complement. However, the following further improvements hold for both encodings:
Having computed the matrix whose rows we want to sum up, we can apply a $\log(n+m)$-depth circuit for adding the $n+m$ rows. It is noteworthy that we can save computation power by modifying the addition operation: As can easily be seen, we are always adding rows of different lengths. While the naive approach of padding the right-hand side of the shorter number with 0's and applying normal addition would also work, we can save some effort by copying the excess bits of the longer number and then performing addition on the remaining shorter equal-length parts. Generally, when using this second approach, we only perform an addition of the length of the shorter input, which is an important factor in depth optimization.

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

5.4 Managing Length

By default, each addition and each multiplication increase the bitlength: Addition increases it by 1, whereas multiplication results in a bitlength that is the sum of the two input lengths. When performing several multiplications consecutively, this can easily lead to enormous bitlengths. However, in a scenario where the size of the values can be estimated, there is a way around this. One such scenario is machine learning, where the person working on the data is the person who has the algorithm for building the model and it is a reasonable assumption that some factors of the model are known, e.g. from experience. For example, in the data set we worked with (see Section 6.3.1), the value w_0 always took some value near 10000 no matter what subset of test subjects we chose. In such cases, the service provider who is doing the computations can put a bound on the lengths (i.e., he is certain that the weights will not be larger in absolute value than 2^q for some q). When this is the case, we can reduce the bitlength of the encrypted values to this size $q + 1$ by discarding the excess bits: In Two's Complement, we can delete the most significant bits (which will all be 0 for a positive and 1 for a negative number) until we reach the desired length, whereas for Sign-Magnitude we discard the bits following the MSB (which will all be 0). More specifically, we actually integrated this into our multiplication routine, such that we not only save space, but also effort, as we only compute until we reach the bound in each step. This can be viewed as the inversion of the sign extension operation introduced in Section 3.3.1 and makes the entire algorithm significantly faster, as we have elimninated linear growth in the bitlength.

6 Applications

In this section, we demonstrate the performance increase on two concrete use cases. All computations were done on a virtual machine with 5 GB of RAM running Ubuntu 14.04 LTS (running on a Lenovo Yoga 2 Pro with a Intel i7-4500U processor with 1.8 GHz and 8 GB of RAM with Windows 8.1). Encryptions were done using the second parameter set from the `Test_bootstrapping.cpp` file included in the HElib library.

6.1 Background and Motivation

Generally, Fully Homomorphic Encryption allows the computation of arbitrary functions on encrypted data while keeping the data hidden from the computing party. While FHE does not in principle offer to keep the function private (e.g., if the data and the function belong to the same party, who wishes to have the computation done by a different party with more computing power, like a cloud service provider), it can hide the function that was applied in the following case: If the data belongs to one party and the function belongs to the computing party, then FHE schemes that are "circuit private" guarantee that a ciphertext divulges nothing about the function that was applied to it. Since circuit privacy is often a goal for FHE schemes, it makes sense to extend this requirement to the encoding choices to achieve privacy for the end result. This then means that the data owner learns nothing about the applied function except for what he can derive from the result, and the function owner learns nothing about the data. In this spirit, machine learning has often been cited as an application of Fully Homomorphic Encryption (see Section 2). Machine learning describes a field of research focused on extracting information from data, e.g. in the form of models. In this paper we consider the following scenario: Suppose Alice has a machine learning algorithm which takes data as input and returns a predictive model, and Bob has some data and would like either to obtain a model based on his data, or apply said model to further data (though he does not obtain the model in that case, e.g. allowing the service provider to bill him for each classification of his data). However, Alice does not want to reveal her algorithm for building the model to Bob, and Bob wishes to keep his data secret. With Fully Homomorphic Encryption, Bob could encrypt his (training) data and send it to Alice, who then performs her algorithm on the encrypted data. The output is an encryption of the model, which Alice can apply to new encrypted data instances from Bob and Bob only receives the result of applying the model to his data (first case), or the whole model is sent to Bob (second case), in which case only Bob can decrypt the model. Thus, with an adequately secure Fully Homomorphic Encryption scheme, Alice has learned nothing about Bob's data and Bob has learned nothing about Alice's algorithm except what he can deduce from the result of the evaluation.

In the following, we consider two use cases, one for each of the above scenarios. For the first case, we take up a use case already presented in [21]: the Linear Means Classifier, where we assume that the model has already been built. Alice receives Bob's encrypted data, which she classifies by evaluating a polynomial of degree 2. This use case showcases our new Hybrid Encoding, which performs significantly better in this general case where the results are not bounded.

For the second case, we examine the Perceptron and show how to improve efficiency in evaluating it, showcasing our results regarding choice of encoding and tweaks in multiplication. The Perceptron is a classifier: There are subjects with known classifications which are used to build the model, subjects with known classification to test the model, and the output is the model which can be used to classify future instances with unknown classifications. The Perceptron is an important fundamental algorithm in machine learning upon which many others are built, so being able to efficiently homomorphically evaluate it is mandatory before we can move on to more advanced machine learning algorithms.

The given runtimes are estimates for data encrypted with HElib ([1]), as runtimes are still very large: We measured the time taken for operations like addition and multiplication for different parameters and extrapolated the time it would take to compute the entire function. For example, given the function $f(x_1, x_2, x_3, x_4) = x_1 \cdot x_2 + x_3 \cdot x_4$ on inputs of length n, we would calculate the runtime as that of 2 multiplications of numbers of lengths n plus one addition of numbers of lengths $2n$ (in the unbounded case). We confirmed our computations by actually running the Perceptron for lengths $n = 3$ and $n = 5$ for all three encodings to make sure that our computations reflect reality. However, we would also like to point out that these runtimes depend greatly on the characteristics of HElib: If one used a different encryption scheme that takes longer or shorter to perform bootstrapping, the results would vary greatly. However, our theoretical results are independent of the scheme that was used.

6.2 Homomorphically Evaluating the Linear Means Classifier

In this section, we examine the Linear Means Classifier to showcase the first use case, where the Service Provider retains the encrypted model and the user may send further encrypted data which is then classified by the encrypted model and only the encrypted result is returned to the user.

6.2.1 The Linear Means Classifier

As already implied by its name, the Linear Means classifier classifies data. Like [21], we consider the case where there are two classes, which are determined by the sign of the score function. This score function is a polynomial of degree 2. More concretely, the model consists of a vector $w = (w_1, \ldots, w_l)$ and a constant c, and the data to be classified is a l-dimensional real-valued vector $x = (x_1, \ldots, x_l)$. The score function is then computed as $\langle w, x \rangle + c = w_1x_1 + w_2x_2 + \cdots + w_lx_l + c$, and the sign of the result determines which class the data instance belongs to. As can easily be seen, this is closely related to the classification function of the Perceptron from the next section, where the focus is on determining w and c instead of computing the score function for given (encrypted and thus unknown) values for w and c as we do here.

6.2.2 Performance

Using the Linear Means Classifier, we now examine the effects of using different encodings in the unbounded case (i.e., when the product of two n-bit numbers has length $2n$). To this end, we compute both the effort required in terms of bit operations and depth and the runtime of evaluating the score function for inputs of bitlength 30 for different numbers l of features. As explained above, we computed these runtimes from their components (i.e., the runtime for multiplying two 30-bit numbers without bounds, and the runtime for adding two 60-bit numbers) as the numbers are quite large. The results can be found in Table 3 and also in Figure 2 on page 5.

l	TC+	TC*	SM+	SM*	HE+	HE*	RT$_{TC}$	RT$_{SM}$	RT$_{HE}$
5	43690	13355	43210	16960	27050	10625	322120	368250	251165
10	87380	26710	86420	33920	54100	21250	644240	736500	502330
20	174760	53420	172840	67840	108200	42500	1288480	1473000	1004660
50	436900	133550	432100	169600	270500	106250	3221200	3682500	2511650
250	2184500	667750	2160500	848000	1352500	531250	16106000	18412500	12558250

Table 3: Number of binary additions (TC+, SM+ and HE+) and binary multiplications (TC*, SM* and HE*) for Linear Means Classification with bitlength 30 and l features, as well as runtimes in seconds (RT$_{TC}$, RT$_{SM}$ and RT$_{HE}$). Multiplicative depth is always 352 for TC, 297 for SM, and 326 for HE.

As we can see, using our new Hybrid Encoding significantly improves all aspects except depth, which is about halfway between the other two encodings, which did not matter in our case as we bootstrapped after every multiplication.

6.3 Homomorphically Evaluating the Perceptron

In this section, we examine the first use case where the Perceptron is evaluated to return an encrypted model.

6.3.1 The Perceptron

The Perceptron is an algorithm based on neural networks and basically works by computing a weighted sum of the input traits $x_{k,i}$, which are usually rational numbers, for each subject k and then classifying into one of two classes depending on whether this weighted sum is above a certain threshold or not. In the training phase, the weights are adjusted if the computed classification does not match the known classification $c_k \in \{0, 1\}$ of the training instance. The learning rate η determines by how much the weights change through each such mismatch. A larger η means bigger changes and less stability, whereas for a smaller value, more rounds may be needed, but the weights are more likely to converge. After training, the model can be used to classify future inputs with no known classification. The model consists of the weights, and the threshold can either be predetermined or flexible (and thus part of the model being computed). We will work with the latter approach and for notation reasons include a dummy trait that is always -1, which enables us to compare the scalar product to 0. (Denoting the threshold τ, we have
$\sum\limits_{i=1}^{m} w_i \cdot x_i > \tau \Leftrightarrow \sum\limits_{i=0}^{m} w_i \cdot x_i > 0$ for $x_0 = -1$ and $w_0 = \tau$.)

Note that this is a binary classifier, i.e., it only works with two classes, but the more complex case of several classes can easily be built by running the Perceptron several times for different classes or by having more than one output (i.e., computing several sums and having the targets not be bits, but bitstrings). The exact workings of the Perceptron are presented in Figure 6, and further implementation details can be found below.

We will now discuss the parameters and the dataset used in our actual implementation of the Perceptron. To test our implementation, we used the Pima Indian Dataset [2]. This is a dataset concerning 768 females at least 21 years old which considers 8 different traits and classifies into "has diabetes" or "does not have diabetes". Since the weights for two attributes did not seem to converge at all, we reduced the number of traits down to $m = 6$:

1. Number of times pregnant
2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test
3. Diastolic blood pressure (mm Hg)
4. Body mass index $\frac{\text{weight in kg}}{(\text{height in m})^2}$
5. Diabetes pedigree function
6. Age (years)

We set the learning rate as $\eta = 0.125$ and reserved the last t subjects for performing the testing phase (i.e., use the weights obtained from the model to classify entries with known class which were not involved in the generation of the weights and see how many are correct). This means that we worked with $v = 768 - t$ subjects in the training phase. Note that the testing phase is not carried out bitwise and does not really belong to our encrypted model as it would be carried out client-side, but we performed it to see how different precision values influenced the accuracy of our derived model. We used different precision values and bounds on the length, and these computations were done in unencrypted form, encoded bitwise, as they were only to determine satisfactory parameter values. As it turns out, 20 bits with 6 bits for precision is not enough, whereas both bitlengths 25 (precision 10) and 30 (precision 15) yielded satisfactory results. From previous experiments, we knew that w_0 (i.e., the weight multiplied with $x_0 = -1$) always converged to a number around 10000, so we initialized w_0 as $10000 + r$ where r is a small random number. As already mentioned in section 5.4, we feel that this is a feasible scenario: Suppose Alice has a great algorithm that builds models to classify diabetes occurence according to the above-mentioned attributes, and in designing her model has run it on different population groups. Then we feel that it is feasible for her to know from experience things like the rough range to expect for certain values, like the weight for the dummy input being around 10000. Note that we could live without this assumption, but we would need more rounds, as the weight only changes by a small amount in each weight update and it would take many rounds

```
Phase Training
Input:
    • Training Data $(x_{k,i}, c_k), k = 0, \dots, v-1,$     $i = 1, \dots, m$ of $v$
      subjects with $m$ traits
    • Learning Rate $\eta$
    • Iteration Number $R$
Write inputs as $X$, a $v \times (m+1)$-matrix with first column $-1$, followed by
the inputs $x_k$ row-wise, and $v$-dimensional target vector $\vec{c}$ with entries $c_k$.

    for $0 \leq i \leq m$ do
        $w_i \leftarrow \mathcal{R}_\epsilon$ (Small random number)
    end for
    for $R$ iterations do
        for $0 \leq k \leq v-1$ do
            if $\sum_{i=0}^{m} w_i \cdot x_{k,i} > 0$ then
                Set $y = 1$
            else
                Set $y = 0$
            end if
            Update the weights:
            for $0 \leq i \leq m$ do
                $w_i \leftarrow w_i + \eta \cdot (c_k - y) \cdot x_{k,i}$
            end for
        end for
    end for
Output: $w_i$ for $0 \leq i \leq m$
end Phase
________________________________________________________________
Phase Classification
Input: $w_0, \dots, w_m$ from Training Phase, Vector $x = (x_1, \dots, x_m)$ to be clas-
sified.

    Set $x_0 = -1$
    if $\sum_{i=0}^{m} x_i \cdot w_i > 0$ then
        Set $y = 1$
    else
        Set $y = 0$
    end if
Output: Classification $y$
end Phase
```

Figure 6: The Perceptron

to reach a value as big as 10000. To reduce runtime, in our implementation we chose to precompute the values $\eta \cdot c_k \cdot x_{k,j}$ and $-\eta \cdot x_{k,j}$ for all $k = 0, \dots, v-1$ and all $j = 0, \dots, m$, as these values don't change from round to round.

6.3.2 Performance

We will now examine how the optimizations from Section 5 affect the Perceptron, as shown in Figure 7.

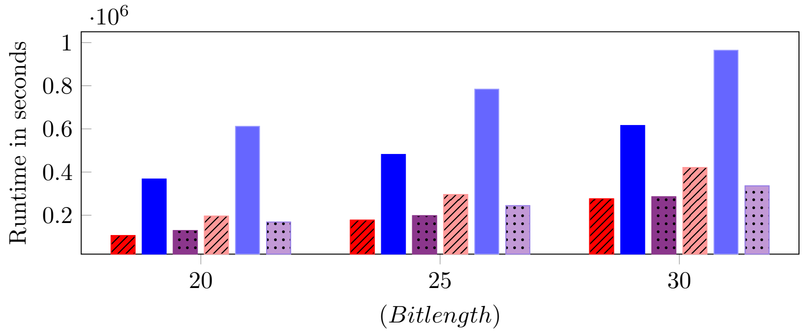


Figure 7: Extrapolated runtimes for one subject for the first round of the encrypted Perceptron for Two's Complement (● (lines) for bounded values, ● (lines) for unbounded values), Sign-Magnitude (● (solid) for bounded values, ● (solid) for unbounded values) and using our new Hybrid Encoding (● (dotted) for bounded values, ● (dotted) for unbounded values).

We can see that bounding the values makes a huge difference, especially since these values are only for the first round and would grow exponentially in further rounds. Sign-Magnitude is consistently the worst choice, and in the unbounded case, Hybrid Encoding is fastest (as already evident from Section 6.2). In the bounded case, however, Two's Complement is fastest, and this makes sense: The fact that we have integrated the bounding into our multiplication procedure and stop computing in each line as soon as the bound is reached negates the sign extension that incurs the slowdown for multiplication in Two's Complement encoding. This means that we expect bounded Two's Complement multiplication to be almost as fast as Sign-Magnitude multiplication, which was confirmed by our experiments. Due to this, there is no efficiency gain through our new encoding in the bounded case, but the graph still illustrates the importance of choosing the right encoding, as Sign-Magnitude is significantly slower here due to its costly addition.

7 Conclusion and Future Work

In conclusion, we have presented a way of working with encrypted rational numbers, to our knowledge being the first to not restrict ourselves to unsigned integers. We have presented a new hybrid encoding technique that vastly improves efficiency for FHE on rational numbers both in theory and for real-world applications like the Linear Means Classifier, and other optimizations that improve efficiency for more complicated functions like the Perceptron. Since our results are independent of the scheme used, they hold with maximum generality and can thus be beneficial for anyone looking to evaluate a function homomorphically. For future research, we believe that this hybrid approach may be transferable to plaintext spaces other than $\{0, 1\}$, although the elementary operations will be considerably more involved. Further, we imagine that it could be beneficial to take a step back from established encodings and come up with a new one from scratch, which could be specially tailored to FHE computations.

Acknowledgements

We would like to thank Thomas Schneider and the anonymous ACNS reviewers for their valuable insights, as well as Shai Halevi for his help with implementation issues.

This work was supported by the Juniorprofessoren-Programm Baden-Württemberg 2013.

References

[1] Helib library: https://github.com/shaih/helib.

[2] Pima dataset: https://archive.ics.uci.edu/ml/datasets/pima+indians+diabetes.

[3] F. Armknecht, C. Boyd, C. Carr, K. Gjøsteen, A. Jäschke, C. A. Reuter, and M. Strand. A guide to fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2015/1192.

[4] F. Armknecht, S. Katzenbeisser, and A. Peter. Group homomorphic encryption: characterizations, impossibility results, and applications. *DCC*, 2013.

[5] F. Armknecht and T. Strufe. An efficient distributed privacy-preserving recommendation system. In *Med-Hoc-Net*, 2011.

[6] L. J. M. Aslett, P. M. Esperança, and C. C. Holmes. Encrypted statistical machine learning: new privacy preserving methods. *CoRR*, abs/1508.06845, 2015.

[7] J. W. Bos, K. E. Lauter, and M. Naehrig. Private predictive analysis on encrypted medical data. *Journal of Biomedical Informatics*, 50, 2014.

[8] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser. Machine learning classification over encrypted data. In *NDSS*, 2015.

[9] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *ECCC*, 18, 2011.

[10] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS*, 2011.

[11] J. H. Cheon, M. Kim, and K. E. Lauter. Homomorphic computation of edit distance. In *FC*, 2015.

[12] H. Chung and M. Kim. Encoding rational numbers for fhe-based applications. *IACR Cryptology ePrint Archive*, 2016/344.

[13] J. Coron, T. Lepoint, and M. Tibouchi. Scale-invariant fully homomorphic encryption over the integers. In *PKC*, 2014.

[14] J. Coron, D. Naccache, and M. Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *EUROCRYPT*, 2012.

[15] A. Costache, N. P. Smart, S. Vivek, and A. Waller. Fixed point arithmetic in SHE scheme. *IACR Cryptology ePrint Archive*, 2016/250.

[16] N. Dowlin, R. Gilad-Bachrach, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing. Manual for using homomorphic encryption for bioinformatics. Technical Report MSR-TR-2015-87, Microsoft Research, 2015.

[17] L. Ducas and D. Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT*, 2015.

[18] C. Gentry. *A fully homomorphic encryption scheme.* PhD thesis, Stanford University, 2009.

[19] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, 2012.

[20] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO*, 2013.

[21] T. Graepel, K. E. Lauter, and M. Naehrig. ML confidential: Machine learning on encrypted data. In *ICISC*, 2012.

[22] W. Henecka, S. Kögl, A. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: tool for automating secure two-party computations. In *CCS*, 2010.

[23] A. Jäschke and F. Armknecht. Accelerating homomorphic computations on rational numbers. In *ACNS*, 2016.

[24] M. Naehrig, K. E. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *CCSW*, 2011.

[25] A. Sadeghi and T. Schneider. Generalized universal circuits for secure evaluation of private functions with application to data classification. In *ICISC*, 2008.

[26] N. P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *PKC*, 2010.

[27] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *SP*, 2015.

[28] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *EUROCRYPT*, 2010.

[29] D. J. Wu, T. Feng, M. Naehrig, and K. E. Lauter. Privately evaluating decision trees and random forests. *IACR Cryptology ePrint Archive*, 2015/386.

A How to Get From Pseudocode to a Boolean Circuit

Since literature on Fully Homomorphic Encryption often talks about transforming a function into a boolean circuit, we have decided to include this appendix as an example, as it might not be clear to the reader how to make this transformation. To this end, we will quickly show how to express the addition of Two's Complement numbers and the comparison of Sign-Magnitude numbers as polynomials in their input bits, which is equivalent to the corresponding circuit, and give a short complexity analysis. To evaluate the function in encrypted form, one now merely has to exchange every $+$ with the corresponding addition operation on the ciphertexts and every $\cdot$ with the multiplication operation on the ciphertexts. Since we are working with bits, $+$ and $\cdot$ are the operations of $\mathbb{Z}_2$. We chose these specific functions because addition is the computation of a value from two other values and thus an example of an algebraic function, whereas comparison returns only one bit after navigating a number of if-then decisions and thus shows how to represent logical functions.

A.1 Binary Addition (Two's Complement)

Suppose we are adding two n-bit Two's Complement numbers a and b, where $a = a_{n-1}a_{n-2}\dots\ \dots a_1a_0$ and $b = b_{n-1}b_{n-2}\dots b_1b_0$. We first extend both strings by one bit (as described in Section 3.3.1) to obtain $(n+1)$-bit strings $a = a_na_{n-1}\dots a_1a_0$ and $b = b_nb_{n-1}\dots b_1b_0$ with $a_n = a_{n-1}$ and $b_n = b_{n-1}$. We will denote the output with $c = c_nc_{n-1}\dots c_1c_0$ and the carry bits we use with r_i. Then the output bits have the form $c_i = a_i + b_i + r_i$ where $r_0 = 0$. Thus, the real challenge is expressing r_i as a polynomial for $1 \leq i \leq n$. Since the carry bit r_i is 1 when at least two out of the three values $\{a_{i-1}, b_{i-1}, r_{i-1}\}$ are 1, this is

the majority function, which can be expressed as
$r_i = (a_{i-1} + b_{i-1}) \cdot (r_{i-1} + a_{i-1}) + a_{i-1}$ for $2 \leq i \leq n-1$ with $r_0 = 0$ and $r_1 = a_0 \cdot b_0$.
This is not the naive formula, but instead one that results in only a single multiplication. As can easily be seen, the above equation evaluates to 1 exactly when at least two of the values are 1. Thus, we have found a way to recursively define r_i and thus addition as a whole through $c_i = a_i + b_i + r_i$.

Note that in order to save memory when implementing, we do not need to make a vector of carry bits: Indeed, r_{i-1} is only used for computing c_{i-1} and r_i, so it suffices to keep the value in one variable r and in each round (i.e., for each i) setting $r = (a_{i-1} \cdot b_{i-1}) + (r \cdot (a_{i-1} + b_{i-1}))$ and $c_i = a_i + b_i + r$.

In terms of effort, we can easily derive the formulas given in Section 4.2: We have $c_0 = a_0 + b_0$ (because $r_0 = 0$) and then $c_i = a_i + b_i + r_i$ for $i = 1 \dots n$, so we get $1 + 2n$ additions from this formula. From the computation of r_i, we get 0 additions for $r_0 = 0$ and $r_1 = a_0 \cdot b_0$, and 3 additions for $i = 2, \dots, n$, yielding $3 \cdot (n-1)$ further additions. Adding these numbers, we get a total of $2n + 1 + 3n - 3 = 5n - 2$ binary additions.

Likewise, we get 0 multiplications from $c_i = a_i + b_i + r_i$ and $r_0 = 0$, 1 multiplication from $r_1 = a_0 \cdot b_0$, and 1 multiplication from the computation of r_i for $i = 2, \dots, n$, leaving us with $1 + 1 + (n-2) = n$ binary multiplications.

Lastly, for the multiplicative depth, we see that r_1 has depth 1, and then depth increases by 1 for $i = 2, \dots, n$ because r_i is multiplied with another value $(a_{i-1} + b_{i-1})$. Note that for depth, we are only interested in the term with the highest amount of consecutive multiplications, so we ignore the other terms we would get by expanding the formula for r_i, since they have less depth. We obtain a total depth of $1 + (n-1) = n$.

A.2 Comparison (Sign-Magnitude)

When computing a function where the operation to be performed depends on a certain condition, the idea is the following: Compute the boolean value of whether the condition is met, and multiply each bit of the output with this boolean value (through a function denoted `OneBitMult`$(a_{n-1} \dots a_1 a_0, b) = (a_{n-1} \cdot b) \dots (a_1 \cdot b)(a_0 \cdot b)$). This results in a string of 0's if the condition is not met, and the original result string if it is. If the conditions are phrased to be mutually exclusive, we can just add the result for each case component-wise to obtain the end result, as exactly one case-result will be non-zero.

To illustrate, we examine the comparison of two numbers $a = a_{n-1} \dots a_1 a_0$ and $b = b_{n-1} \dots b_1 b_0$ in Sign-Magnitude encoding. First, we need to identify all the cases that should return 1, i.e., where $a \leq b$:

i. $a_{n-1} = 1$ and $b_{n-1} = 0$ (a negative and b positive)

ii. $a_{n-1} = b_{n-1} = 1$ and $|a| \geq |b|$ (both negative, a bigger absolute value)

iii. $a_{n-1} = b_{n-1} = 0$ and $|a| \leq |b|$ (both positive, a smaller absolute value)

iv. $a = 000\dots00$ and $b = 100\dots00$ (two representations of 0)

Note that $|a| \geq |b|$ if $a_{n-2} = 1$ and $b_{n-2} = 0$ or $a_{n-2} = b_{n-2}$ and $a_{n-3} = 1$ and $b_{n-3} = 0$ or ..., i.e. if at the highest index where $a_{n-2}\dots a_1a_0$ and $b = b_{n-2}\dots b_1b_0$ differ, a has the value 1. Likewise, $|a| \leq |b|$ if at the highest index where they differ, a has the value 0. Since the above 4 cases are mutually exclusive, we can express each as a polynomial and add them together, since at most one of them will evaluate to 1. We express as a polynomial by writing $+$ for OR (technically XOR, which is why it is important to make sure that the cases are mutually exclusive) and $\cdot$ for AND:

i. $p_1 = a_{n-1} \cdot (b_{n-1} + 1)$

ii. $p_2 = (a_{n-1} \cdot b_{n-1}) \cdot (a_{n-2} \cdot (b_{n-2} + 1)$
$\quad + (a_{n-2} + b_{n-2} + 1) \cdot ((a_{n-3} \cdot (b_{n-3} + 1) + (\dots))),$
or alternatively written:

   1. $p_2 \leftarrow a_0 \cdot (b_0 + 1) + (a_0 + (b_0 + 1))$
   2. For $1 \leq i \leq n-2$:
      $p_2 \leftarrow (p_2 \cdot (a_i + b_i + 1)) + (a_i \cdot (b_i + 1))$
   3. $p_2 \leftarrow p_2 \cdot a_{n-1} \cdot b_{n-1}$

iii. $p_3 = ((a_{n-1} + 1) \cdot (b_{n-1} + 1)) \cdot ((a_{n-2} + 1) \cdot b_{n-2}$
$\quad + (a_{n-2} + b_{n-2} + 1) \cdot (((a_{n-3} + 1) \cdot b_{n-3} + (\dots))),$
or alternatively:

   1. $p_3 \leftarrow ((a_0 + 1) \cdot b_0) + (a_0 + b_0 + 1)$
   2. For $1 \leq i \leq n-2$:
      $p_3 \leftarrow (p_3 \cdot (a_i + b_i + 1)) + ((a_i + 1) \cdot b_i)$
   3. $p_3 \leftarrow p_3 \cdot (a_{n-1} + 1) \cdot (b_{n-1} + 1)$

iv. $p_4 = (a_{n-1} + 1) \cdot (a_{n-2} + 1) \cdot \dots (a_1 + 1) \cdot (a_0 + 1)$
$\quad \cdot b_{n-1} \cdot (b_{n-2} + 1) \cdot \dots (b_1 + 1) \cdot (b_0 + 1)$

Thus, the polynomial $p = p_1 + p_2 + p_3 + p_4$ is the function that returns 1 exactly when $a \leq b$.

If we want to compute the effort from Section 4.1, we again just count the number of additions and multiplications: We get 1 addition and 1 multiplication from p_1, $4 + (n-2) \cdot 4$ additions and $1 + (n-2) \cdot 2 + 2$ multiplications from p_2, $4+(n-2)\cdot4+2$ additions and $1+(n-2)\cdot2+2$ multiplications from p_3, $n+(n-1)$ additions and $(n-1)+n$ multiplications from p_4 and 3 additions from summing up the 4 polynomials. In total, we get $1+4+(n-2)\cdot4+4+(n-2)\cdot4+2+n+(n-1)+3 = 10\cdot n-3$ additions and $1+1+(n-2)\cdot2+2+1+(n-2)\cdot2+2+(n-1)+n = 6n-2$ binary multiplications. Regarding multiplicative depth, we can see that the most consecutive multiplications occur in p_4, where we get a depth of $2n-1$.