Universität Siegen

Naturwissenschaftlich-Technische Fakultät

Bachelor's Thesis

for acquiring the academic degree of Bachelor of Science Computer Science

The State of Homomorphic Encryption

Coach	Prof. Bernhard Esslinger
	Dr. Doris Behrendt
First examiner	Prof. Bernhard Esslinger
Second examiner	Prof. Dr. Roland Wismüller

Author	Michael Heep

Submitted	15.06.2023
Update	05.02.2024

Kurzzusammenfassung

Das Ziel dieser Bachelorarbeit ist es, den aktuellen Stand homomorpher Verschlüsselung aufzuzeigen. Dazu wird erklärt, was homomorphe Verschlüsselung ist und welche Rolle die sogenannte volle homomorphe Verschlüsselung heutzutage spielt. Dabei wird auch auf die Herausforderungen bezüglich der Benutzung solcher Verschlüsselungsverfahren hingewiesen, sowie auf die Schwiergkeit die Theorie hinter denselben zu erklären.

Es wird ein Anwendungsfall vorgestellt, in dem mit Hilfe von homomorpher Verschlüsselung eine geheime Umfrage ermöglicht wird. Anhand der Implementierung dieses Anwendungsfalles wird gezeigt, wie das Rechnen auf verschlüsselten Daten in der Praxis aussieht.

Homomorphe Verschlüsselung macht sowohl in der Theorie als auch in der Praxis rasche Fortschritte, wie die kontinuierliche Entwicklung neuer Verfahren und ihre Implementierung in gängigen Bibliotheken zeigt. Allerdings schränken Herausforderungen wie die Notwendigkeit umfangreicher theoretischer Kenntnisse und die hohen Rechenanforderungen die praktische Relevanz der vollständig homomorphen Verschlüsselung immer noch stark ein.

Abstract

The goal of this bachelor's thesis is to show the current state of homomorphic encryption (HE). The meaning of homomorphic encryption and the role of fully homomorphic encryption (FHE) are explained. The challenges related to the use of such encryption schemes are also addressed, as well as the difficulty of explaining the theory behind them.

A use case is presented in which homomorphic encryption is used to enable a secret poll. By implementing this use case, it is demonstrated how computation on encrypted data looks in practice.

Homomorphic encryption is advancing rapidly in both theory and practice, as indicated by the continuous development of new schemes and their implementation in popular libraries. However, challenges such as the need for extensive theoretical knowledge and the high computational requirements still severely limit the practical relevance of fully homomorphic encryption.

Contents

1 Introduction

This thesis is about the current state of homomorphic encryption (HE). HE is a special variant of public-key cryptography that allows certain computations on encrypted data. A recent development regarding HE is the advent of applicable fully homomorphic encryption (FHE). [23, 14, 11, 19, 20, 28, 29, 31, 15, 8] In contrast to homomorphic encryption (HE), fully homomorphic encryption (FHE) enables almost unlimited computations on encrypted data.

The relevance of HE in cryptography is confirmed by numerous presentations at major conferences all around the world in recent years. For example, HE is a topic at nearly every one of the IACR's[1] annual events: Eurocrypt [42, 25, 45, 37, 41, 33], Crypto [43, 2, 59], Asiacrypt [46, 10, 16], Public Key Cryptography (PKC) [7, 40], Cryptographic Hardware and Embedded Systems (CHES) [36], Theory of Cryptography Conference (TCC). [3] After Craig Gentry's breakthrough paper in 2009 showed that FHE is theoretically possible [26], there have been more and more publications on HE every year. To illustrate the number of publications over the years, see Fig. 1.[2]

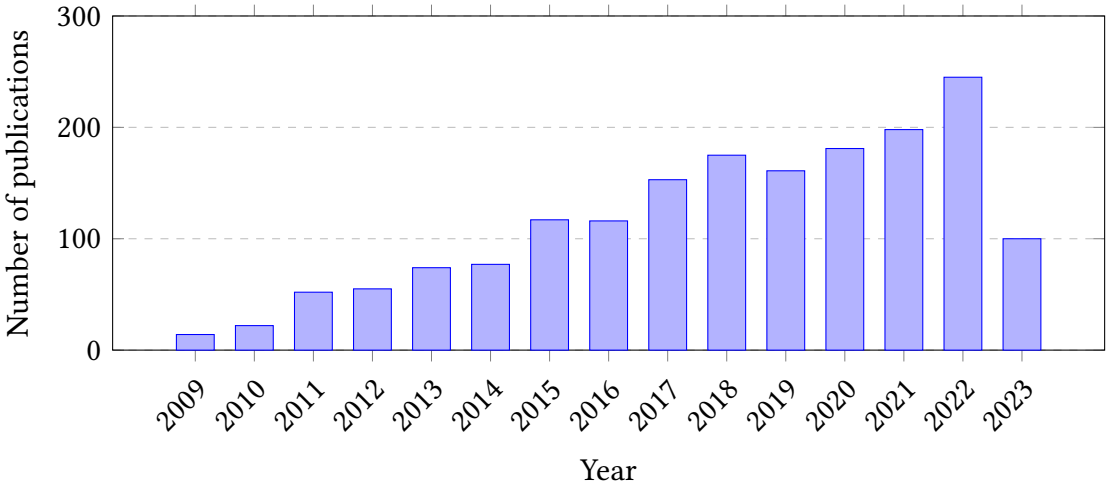

Figure 1: Number of publications on homomorphic encryption per year

Some applications have already been successfully implemented using current FHE schemes. The implementation of a „multinomial logistic regression" is an example of a secure service that works with encrypted sensitive client data. This provides a solution to the problem of „secure genotype imputation". [51, p. 5] Regarding multi-

[1] International Association for Cryptologic Research: https://iacr.org
[2] https://dblp.org/search?q=homomorphic%20encryption (May 27, 2023).

party communication (MPC), a passively-secure oblivious linear function evaluation (OLE) protocol was implemented with FHE. „This protocol generalizes oblivious transfer to linear functions, and its [...] implementation [...] is able to evaluate more than 1 million OLEs per second over the ring $\mathbb{Z}_m$, for a 120-bit m on standard hardware.“ [51, p. 5]

Another very interesting use case is the training and evaluation of different machine learning methods. Examples are generalized linear models on distributed datasets [24] or feed-forward neural networks. [61]

Circumstances that currently limit the practical use of FHE are: 1. The very large size of ciphertexts leads to extensive memory load. [52, p. 32] 2. The runtime of computations on encrypted data is several orders of magnitude slower than the same computations on unencrypted data, depending on the use case.[3] 3. Setting reasonable starting parameters for an FHE scheme is extremely complicated, but necessary to efficiently perform desired calculations at a required level of security. This represents a very high entry point for beginners and non experts.

Outline Chapter 2 describes the basic theory of homomorphic encryption (HE) and fully homomorphic encryption (FHE) (Sections 2.1 to 2.2) and mentions their different variants and generations (Section 2.3). It also covers in Section 2.4 the difficulties in handling FHE schemes, e.g., setting proper parameters. In Section 2.5 general algorithms belonging to FHE schemes are given. Section 2.6 presents a special improvement of some FHE schemes called RNS. In Section 2.7 security aspects are mentioned.

Chapter 3 presents an implementation of a web app using HE. The process of selecting a use case (Section 3.1 on page 38) and a library (Section 3.2 on page 44) is explained. Implementation details are given in Section 3.3 on page 46. In Section 3.4 on page 49 screenshots of the application are presented. Finally, in Section 3.5 on page 54 the integration into CrypTool-Online (CTO) is described.

Chapter 4 presents the findings of this thesis and comes to a conclusion about the state of the FHE.

[3] „On a modification of the MNIST benchmark dataset for handwritten number recognition (consisting of 60 000 training datasets and 10 000 test datasets, each 28×28 grayscale images), the HELib library had a runtime that was a factor of 10^6 slower than running directly on unencrypted data, despite optimizations. [53] The achieved security was 80 bits.“ [52, p. 54][18]

2 Theory

This chapter describes the basics of homomorphic encryption (HE) in Section 2.1. In Section 2.2 on page 14, the construction of fully homomorphic encryption (FHE) schemes is explained. Section 2.3 on page 18 shows the generations and variants of FHE schemes that have emerged in recent years. The difficulties associated with the use of FHE schemes are discussed in Section 2.4 on page 21. In particular, these difficulties are mainly found in the parameter selection, which is very complex and handled very inconsistently in the literature. Afterwards, in Section 2.5 on page 29, the generic algorithms are mentioned, which make up all FHE schemes according to the homomorphic encryption standard. [4] Finally, we briefly address the security of FHE in Section 2.7 on page 37.

2.1 Homomorphic encryption

HE enables the evaluation of functions on encrypted data. This is made possible by the idea that the decryption function of a public-key encryption (PKE) scheme[4] has to be a homomorphism. [22, p. 398]

The concept of a homomorphic decryption function is illustrated below in Section 2.1.1 on the following page using the well-known Rivest-Shamir-Adleman (RSA) cryptosystem. [58] Textbook (i.e., unpadded[5]) RSA is homomorphic in the sense that the multiplication of ciphertexts is connected with the plaintexts. More precisely, the product of two RSA ciphertexts is an encryption of the product of the original plaintexts. While this is always the case for multiplication, for addition this is not true in general. A counter-example is given below.

A **formal definition** of homomorphic encryption can be found in [26]. For our needs, we now formulate a somewhat reduced version that does not need the whole mathematical apparatus. So, first we have an operation „op“ on some plaintext set $\{m_1, m_2, \dots\}$ and some corresponding operation $\widetilde{\text{op}}$ on the ciphertext set $\{c_1, c_2, \dots\}$. Furthermore, let $I = \{i_1, i_2, \dots, i_k\}$ be an index set such that $m_{i_1}, m_{i_2}, \dots m_{i_k}$ are contained in the plaintext space, then **homomorphic encryption means:**

[4] HE schemes based on symmetric encryption schemes are also possible. [27, p. 98] For example [19, pp. 1-2] and [44].
[5] What padding is, can be found e.g. in [63]

$$\mathrm{op}(m_{i_1}, \ldots, m_{i_k}) = \mathrm{dec}(\widetilde{\mathrm{op}}(\mathrm{enc}(m_{i_1}, \ldots, m_{i_k}))) \tag{1}$$

or

$$\mathrm{enc}(\mathrm{op}(m_i)) = \widetilde{\mathrm{op}}(\mathrm{enc}(m_i)) \tag{2}$$

Note that usually the operation is carried out only on two plaintexts but in the general case depending on the specific operation more than two operands are possible. Also one can think of op being a polynomial or some other function.

There are two special cases, namely addition and multiplication. In the first case the encryption function enc is called additively homomorphic, in the second case the function is called multiplicatively homomorphic. Also, for both special cases, the operation in the ciphertext space and the plaintext space is the same, which does not have to be the case in general, see Table 1 on page 14.

additively homomorphic:
$\mathrm{enc}(m_1 + m_2) \quad = \quad \mathrm{enc}(m_1) + \mathrm{enc}(m_2) \qquad \mathrm{op} = \widetilde{\mathrm{op}} = \text{addition}$

multiplicatively homomorphic:
$\mathrm{enc}(m_1 \cdot m_2) \quad = \quad \mathrm{enc}(m_1) \cdot \mathrm{enc}(m_2) \qquad \mathrm{op} = \widetilde{\mathrm{op}} = \text{multiplication}$

Now we look at RSA, an example of the second special case where „op“ is the multiplication mod n and the operations are identical $\widetilde{op} = op$.

2.1.1 Example: RSA

The following is an informal description of the textbook RSA cryptosystem. [12, pp. 169-172] The public key (encryption key) consists of two positive integers (e, n), whereas the private key (decryption key, secret key) consists of the two positive integers (d, n). The keys are obtained as follows:

Pick two different odd very large prime numbers p, q at random and multiply them to have $n = p \cdot q$. Now calculate the totient of n (i.e., Euler's totient function or Euler's phi function), which, in the case of the product of two primes, is $\phi(n) = (p-1) \cdot (q-1)$. Now, as part of the secret key, choose a large random integer[6] d that is coprime to

[6] Today, the public key is usually set to $e = 2^{16} + 1 = 65537$, and the secret key d is calculated from it. [12, p. 176]

$\phi(n)$, i.e., $\gcd(d, \phi(n)) = 1$. Here gcd stands for *greatest common divisor*. The next step is to calculate the first integer of the public key, which is the multiplicative inverse of d modulo $\phi(n)$. This can efficiently be done using the extended Euclidean algorithm. The result is summarized in the following equation:

$$e \cdot d \equiv 1 \bmod \phi(n)$$

This construction allows for creation of an encryption function *enc* and a decryption function *dec*. The following applies to these functions:

$$\forall m \in \mathbb{Z}/n\mathbb{Z} \,:\, dec(enc(m)) = m \text{ and } enc(dec(m)) = m.$$

To encrypt a message m, calculate

$$c \equiv enc(m) \equiv m^e \bmod n,$$

and to decrypt a ciphertext c, calculate

$$m \equiv dec(c) \equiv c^d \bmod n.$$

RSA is always multiplicatively homomorphic

Let (e, n) be the public key of an RSA cryptosystem and let $m_1, m_2 \in \mathbb{Z}/n\mathbb{Z}$ be two messages. Then

$$\begin{aligned} enc(m_1) \cdot enc(m_2) &\equiv (m_1^e \bmod n) \cdot (m_2^e \bmod n) \\ &\equiv m_1^e \cdot m_2^e \bmod n \\ &\equiv (m_1 \cdot m_2)^e \bmod n \\ &\equiv enc(m_1 \cdot m_2). \end{aligned}$$

This shows that the product of two plaintexts can be built before or after encryption.

RSA is not always additively homomorphic

The following example shows, that under RSA encryption the addition of ciphertexts *can* lead to an encryption of the sum of the plaintexts, but in general this is not the case.

Let the public key be $(e, n) = (5, 35)$. These are valid parameters, because

$$\phi(n) = \phi(35) = \phi(5 \cdot 7) = \phi(5) \cdot \phi(7) = 4 \cdot 6 = 24$$

and

$$\gcd(e, \phi(n)) = \gcd(5, 24) = 1.$$

Now, let $m_1 = 5, m_2 = 10$ be two messages. The addition of the ciphertexts of these messages yields

$$\begin{aligned} enc(m_1) + enc(m_2) &\equiv enc(5) + enc(10) \\ &\equiv (5^5 \bmod 35) + (10^5 \bmod 35) \\ &\equiv (10 \bmod 35) + (5 \bmod 35) \\ &\equiv 15 \bmod 35 \end{aligned}$$

and

$$\begin{aligned} enc(m_1 + m_2) &\equiv enc(5 + 10) \\ &\equiv enc(15) \\ &\equiv 15^5 \bmod 35 \\ &\equiv 15 \bmod 35. \end{aligned}$$

Clearly, in this case $enc(m_1) + enc(m_2) = enc(m_1 + m_2)$. Now, we look at the same computation but with other summands:

$$\begin{aligned} enc(m_1) + enc(m_2) &\equiv enc(4) + enc(12) \\ &\equiv (4^5 \bmod 35) + (12^5 \bmod 35) \\ &\equiv (9 \bmod 35) + (17 \bmod 35) \\ &\equiv 26 \bmod 35 \end{aligned}$$

and

$$\begin{aligned} enc(m_1 + m_2) &\equiv enc(4 + 12) \\ &\equiv enc(16) \\ &\equiv 16^5 \bmod 35 \\ &\equiv 11 \bmod 35. \end{aligned}$$

Obviously, now $enc(m_1) + enc(m_2) \neq enc(m_1 + m_2)$.

Conclusion This example has shown that RSA has homomorphic properties. However, this is only true for textbook, i.e., unpadded RSA and it is only multiplicatively homomorphic. In contrast, RSA is not additively homomorphic in general. Such schemes, which have only one homomorphic property are also called partially homomorphic encryption (PHE) schemes. [52, p. 65]

2.1.2 Other partially homomorphic cryptosystems

There are other public-key cryptosystems that are also homomorphic in one operation. An example where this operation is the addition is the Paillier cryptosystem. [54] It is a probabilistic cryptosystem which is based on the so-called composite residuosity class problem. The theory behind this cryptosystem is out of scope of this thesis and thus left out. However, the homomorphic properties of the Paillier cryptosystem are shown below. [22, pp. 398-399]

The Paillier cryptosystem

The key generation in the Paillier cryptosystem works as follows: The public key n is a valid RSA modulus $n = p \cdot q$ with two different odd very large prime numbers p and q. The private key consists of $\lambda = \text{lcm}(p-1, q-1)$. Here, lcm stands for *least common multiple*.

For each message $m \in \mathbb{Z}/n\mathbb{Z}$ that is to be encrypted, randomly choose an $r \in \mathbb{Z}/n\mathbb{Z}$. To encrypt m with the public key n, calculate

$$c = enc(m, r) = (n+1)^m \cdot r^n \bmod n^2.$$

To decrypt a message $c \in \mathbb{Z}^*_{n^2}$, first calculate

$$S = c^\lambda \bmod n^2$$

and

$$T = \phi(n)^{-1} \bmod n^2.$$

The actual decryption then is

$$m = dec(c) = (S-1)/n \cdot T \bmod n.$$

To see the homomorphic property of this cryptosystem, consider the encryption function E and the decryption function D. Let n be the public key and let $m_1, m_2 \in \mathbb{Z}/n\mathbb{Z}$ be two messages. Also, let $g := n + 1$ for simplicity. The corresponding ciphertexts then are

$$c_1 = g^{m_1} \cdot r_1^n \bmod n^2$$

and

$$c_2 = g^{m_2} \cdot r_2^n \bmod n^2.$$

Now the homomorphic property can be shown as follows:

$$\begin{aligned} enc(m_1, r_1) \cdot enc(m_2, r_2) &\equiv (g^{m_1} \cdot r_1^n \bmod n^2) \cdot (g^{m_2} \cdot r_2^n \bmod n^2) \\ &\equiv g^{m_1+m_2} \cdot (r_1 \cdot r_2)^n \bmod n^2 \\ &\equiv enc(m_1 + m_2, r_1 \cdot r_2). \end{aligned}$$

As you can see, that the operation applied to the ciphertext is not always the same operation carried to the plaintext. While the operation remained the same in RSA, where the product of ciphertexts yields the encrypted product of the plaintexts, in the Paillier cryptosystem a multiplication of ciphertexts yields the encrypted sum of the plaintexts.

2.1.3 Summary

This section has shown that there are already cryptosystems that have homomorphic properties. However, the possible calculations are limited to one type of operation, making them PHE schemes.. For cryptosystems such as RSA and Paillier, the number of calculations that can be performed sequentially is always unbounded. This means, that arbitrarily many of these operations can be performed without loosing any information.

There are other cryptosystems that are partially homomorphic. Some of them are briefly presented in Table 1 on the following page. In these examples, let c_1 and c_2 be the encryption of m_1 and m_2, respectively, under the corresponding cryptosystem. The column *Ciphertext operation* shows what operation is made on the two ciphertexts, and the column *Result on the plaintext* shows the resulting effect this has on the underlying plaintexts. Often, the plaintext operations are modular additions or modular multiplications. See the cryptosystem citations and [52, pp. 17, 65] for details. An example of how to read a row in Table 1 on the next page is: In the Goldwasser-Micali cryptosys-

tem, if you have two ciphertexts $c_1 = enc(m_1)$ and $c_2 = enc(m_2)$[7], then the product of these ciphertexts is an encryption of the exclusive or (XOR) of the plaintexts m_1 and m_2.

Cryptosystem	Ciphertext operation	Result on the plaintext
RSA [58]	$c_1 \cdot c_2$	$m_1 \cdot m_2$
Paillier [54]	$c_1 \cdot c_2$	$m_1 + m_2$
Regev [56]	$c_1 + c_2$	$m_1 + m_2$
Benaloh [9]	$c_1 \cdot c_2$	$m_1 + m_2$
ElGamal [21]	$c_1 \cdot c_2$	$m_1 \cdot m_2$
Goldwasser-Micali [32]	$c_1 \cdot c_2$	$m_1 \oplus m_2$(XOR)

Table 1: Homomorphic properties of some partially homomorphic encryption schemes

2.2 Fully homomorphic encryption

The question that naturally arises from the existence of PHE schemes is: Can there be a cryptosystem that is homomorphic in more than one operation? This problem was proposed as early as 1978 in the paper *On Data Banks and Privacy Homomorphisms* by Rivest et al. [57] The paper is from the same authors who presented the RSA cryptosystem in the same year. But it took about 40 years to find the first answer to this question. In 2009, Craig Gentry showed in [26], that it is possible to construct a so-called fully homomorphic encryption (FHE) scheme.

According to Gentry, *fully homomorphic* is defined as follows: „Given ciphertexts that encrypt $\pi_1, \ldots, \pi_t$, fully homomorphic encryption should allow [...] to put out a ciphertext that encrypts $f(\pi_1, \ldots, \pi_t)$ for any desired function f, as long as that function can be efficiently computed. No information about $\pi_1, \ldots, \pi_t$ or $f(\pi_1, \ldots, \pi_t)$, or any intermediate plaintext values, should leak; the inputs, output and intermediate values are always encrypted.“ [26, p. 5]

The solution found by Gentry contains on the one hand a blueprint to design FHE schemes and on the other hand a concrete example of such a scheme. This scheme is based on ideal lattices and a variant of the closest vector problem (CVP). The theory behind this cryptosystem is by far too complex to be explained in detail in this bachelor's thesis. Therefore, only the superficial blueprint of how to construct an FHE scheme is presented below.

[7] Plaintexts in the Goldwasser-Micali cryptosystem are single bits ($m_1, m_2 \in \{0, 1\}$).

After explaining the blueprint of how to construct an FHE scheme, the question is discussed which functions are homomorphically computable with it.

2.2.1 Gentry's blueprint

The approach consists of three steps, which are described below. [26, 23, 8] Based on Gentry's work, terms such as *squashing a circuit* and *bootstrapping an encryption scheme* have been established.

1. Construct a somewhat homomorphic encryption scheme

The first step is to construct a so-called somewhat homomorphic encryption (SHE) scheme. These type of schemes allow to perform a limited number of computations on the ciphertexts in the form of addition and multiplication. The reason for this limitation is that each ciphertext contains an inherent *noise* parameter, which grows with every homomorphic operation. Especially homomorphic multiplications let the noise grow fast in comparison to homomorphic additions. After too many operations[8], the noise will eventually become too large and the ciphertext can no longer be decrypted successfully.

In Gentry's work, an SHE scheme based on ideal lattices is presented. The security of the scheme is based on a decisional version of the bounded distance decoding problem (BDDP) which is a variant of the CVP. [26, pp. 18, 65]

2. Squashing: Simplify the decryption circuit

The next step is to simplify the decryption algorithm of the SHE scheme, which Gentry called *squashing* the decryption circuit.[9] Specifically, this means lowering the multiplicative depth[10] of it as much as possible. The goal is that the decryption circuit has less depth than the scheme can handle. Only then the next step of the blueprint is possible.

Gentry's idea to squash the decryption circuit of his SHE scheme was to „place a hint about the secret key inside the public key. This hint is not enough to decrypt a cipher-

[8] This limit depends on the scheme that is used and on certain parameters that were selected.

[9] A circuit is a representation of an algorithm that uses (in our case) `AND`, `OR`, and `NOT` gates.

[10] The multiplicative depth of an algorithm, a function or a circuit is the number of sequential multiplications (resp. `AND` gates) in it. For example: The multiplicative depth of $x_1x_2x_3x_4$ is 4 and the multiplicative depth of $x_1x_2 + x_3x_4$ is 2.

text output by the original scheme, but it can be used to ‚process' the ciphertext – i.e., construct a new ciphertext (that encrypts the same thing[11]) that can be decrypted by a very shallow circuit." [26, p. 3] The security of this concept is based on the (decision) sparse subset sum problem (SSSP). For details, see [26, pp. 98, 104].

3. Bootstrapping: Evaluate the decryption circuit homomorphically

The last step is the core of Gentry's construction: The homomorphic evaluation of the simplified decryption function. Gentry has shown that this technique, which he calls *bootstrapping*, reduces the noise of a ciphertext. In particular, one could apply the bootstrapping procedure before each homomorphic operation to guarantee that the result has a low enough noise level to allow correct decryption. In this way, it is theoretically possible to perform an unbounded number of additions and multiplications on encrypted data, giving fully homomorphic encryption (FHE).

2.2.2 Gentry's FHE scheme and its practical drawbacks

The problem with Gentry's initial construction was its performance. The ciphertexts only encrypt a single bit and the homomorphic evaluations took a lot of time. „If one wants 2^λ security [...] the required computation per gate is quasi-linear in λ^6." [26, p. 20] There have been many attempts at implementing schemes, „but none of them comes even close to being practical." [23, p. 2] In [30], C. Gentry and S. Halevi manage „to execute one Advanced Encryption Standard (AES) encryption homomorphically in eight days using a massive amount (tens of GBs) of RAM." [23, p. 2]

2.2.3 What computations are possible with FHE

While the desire is to be able to compute every function on encrypted data, even Gentry's breakthrough did not really make this possible. But this is only because the theoretical limits are the same as those of a „normal computer". For example, neither a computer nor any FHE scheme is capable of performing an infinite number of calculations. But that would be necessary to be able to calculate *all functions*. As an example, the exponential function $\exp(x)$ is defined by the infinite power series

$$\exp(x) := \sum_{k=0}^{\infty} \frac{x^k}{k!}.$$

[11] I.e., a new encryption of the same underlying plaintext.

This type of function can only be calculated by computers using approximation functions that use polynomials with a finite number of terms. Approximate functions only return a result to a certain accuracy.

One of Gentry's ideas was, to reduce the complexity of the functions to be calculated: „How do we measure the complexity of f? Perhaps the most obvious measure is the running time T_f of a Turing machine that computes f. We use a related measure, the size S_f of a Boolean circuit (i.e., the number of AND, OR, and NOT gates) that computes f. Any function that can be computed in T_f steps on a Turing machine can be expressed as a circuit with about T_f gates. More precisely, $S_f < k \cdot T_f \cdot \log T_f$ for some small constant k." [27, p. 99]

The following proof shows that only addition and multiplication suffices to compute the above mentioned gates AND, OR, and NOT. Let $x, y \in \{0, 1\}$ be the inputs to the gates. The following holds [27, p. 99]:

$$\text{AND}(x, y) = x \cdot y, \tag{3}$$

$$\text{OR}(x, y) = 1 - (1 - x) \cdot (1 - y), \tag{4}$$

$$\text{NOT}(x) = 1 - x \tag{5}$$

While Eq. (3) and Eq. (5) are easy to understand, Eq. (4) regarding the OR gate is better illustrated by the truth Table 2.

x	y	$\text{OR}(x, y)$	$(1 - x)$	$(1 - y)$	$(1 - x) \cdot (1 - y)$	$1 - (1 - x) \cdot (1 - y)$
0	0	0	1	1	1	0
0	1	1	1	0	0	1
1	0	1	0	1	0	1
1	1	1	0	0	0	1

Table 2: The formula $1 - (1 - x) \cdot (1 - y)$ decomposed into smaller parts, to better comprehend the correlation with the logic gate OR

Therefore, the basic idea to achieve FHE is to have a cryptosystem that is additively and multiplicatively homomorphic and can perform an indefinite amount of these operations in sequence. With these two operations, every (finite) Boolean circuit can be expressed. Especially, any finite polynomial function can be represented as a sequence of additions and multiplications.

2.3 Generations and variants

Over the years since Gentry's breakthrough, many variants of FHE and SHE schemes have been proposed. Retrospectively, these variants can be categorized in three generations, with each generation representing a conceptual simplification of the schemes. [65, 52] An overview of these schemes is presented on a timeline in Fig. 2 on page 20 and in Table 3 on the next page.

Apart from the below mentioned differences, all FHE schemes of all generations are based on Gentry's blueprint. The differences are to be found, for example, in the underlying problems on which the security of each scheme is based on and the different computational models[12], i.e., on which domain[13] computations are possible.

Most schemes are named after the authors of the papers describing it. Schemes that are used in today's libraries are often abbreviated by the initials of their authors.

Again (like with Gentry's ideal lattice in Section 2.2 on page 14), details of the implementation of each single scheme are omitted. On the one hand, it simply would be too much. On the other hand, it would require an extensive theoretical background that cannot be fathomed in this thesis.

1st Generation

The first generation begins with Gentry's initial construction of 2009. [26] Its security is based on the BDDP and the SSSP. An example of a first generation scheme whose security is based on a different assumption is the Dijk-Gentry-Halevi-Vaikuntanathan (DGHV) scheme (approximate gcd (AGCD), SSSP). [19] The DGHV scheme is also not based on ideal lattices but on simple integer arithmetic. Both schemes encrypt only single Boolean values in their ciphertexts.

2nd Generation

Two of the main schemes of the second generation, that are also offered in today's libraries, are the Brakerski-Gentry-Vaikuntanathan (BGV) scheme [11] and the Brakers-

[12] Computational models can be, for example, modular arithmetic, Boolean arithmetic or floating-point arithmetic

[13] Domains can be, for example, bits, integers, or real numbers.

Gen.	Authors	Underlying problems	Computational model	Year	Ref.
1st	Gentry	BDDP, SSSP	Boolean arithmetic	2009	[26]
	Gentry, Halevi	BDDP, SSSP	Boolean arithmetic	2010	[29]
	DGHV	AGCD, SSSP	Boolean arithmetic	2010	[19]
2nd	BGV	LWE/RLWE	modular arithmetic	2011	[26]
	BFV		modular arithmetic	2012	[29]
	CKKS		floating-point arithmetic	2017	[14]
3rd	GSW	LWE/RLWE	Boolean arithmetic	2013	[31]
	FHEW		Boolean arithmetic	2014	[20]
	TFHE		Boolean arithmetic	2016	[15]

Table 3: Overview of some FHE schemes grouped in three generations [65, 52]

ki/Fan-Vercauteren (BFV) scheme. [23] From the second generation onwards, every FHE scheme relies either on the learning with errors (LWE) or the ring learning with errors (RLWE) assumption. The BGV and BFV schemes provide modular arithmetic over the integers.

Another scheme in this generation is the Cheon-Kim-Kim-Song (CKKS) scheme. [14] Its major difference compared to all other schemes is that it operates on floating-point numbers. While other schemes only encrypt single bits or integers, CKKS can handle other tasks like machine learning to be implemented homomorphically.

In general, second generation schemes are much more efficient than first generation schemes. The efficiency of an FHE scheme can be determined, for example, by the so-called *per-gate computation overhead*. This value is „defined as the ratio between the time it takes to compute a circuit homomorphically to the time it takes to compute it in the clear." [11, p. 1] Given the security parameter λ, the overhead for first generation schemes is in $\tilde{\Omega}(\lambda^4)$.[14] The two variants of the BGV scheme (second generation) have per-gate computation overhead in $\tilde{\Omega}(\lambda \cdot L^3)$[15] and $\tilde{\Omega}(\lambda^2)$, respectively. [11, p. 2].

[14] $\tilde{\Omega}$ means Ω with logarithmic factors ignored. For details, see [64, pp. 48, 63].

[15] Here, L denotes the depth of circuits that can be evaluated by the scheme without bootstrapping. This is referred to as *leveled fully homomorphic encryption*. [11, p. 5]

3rd Generation

Examples of third generation schemes are the FHEW scheme [20] and the Fully Homomorphic Encryption over the Torus (TFHE) scheme. [15] They are conceptually similar to second generation schemes, but the implementations are more sophisticated, so that they accomplish slower noise growth during homomorphic evaluations compared to previous schemes. Both schemes build on techniques of the Gentry-Sahai-Waters (GSW) scheme [31], which is also a third generation scheme. These schemes have their strength in computing Boolean gates on encrypted data.

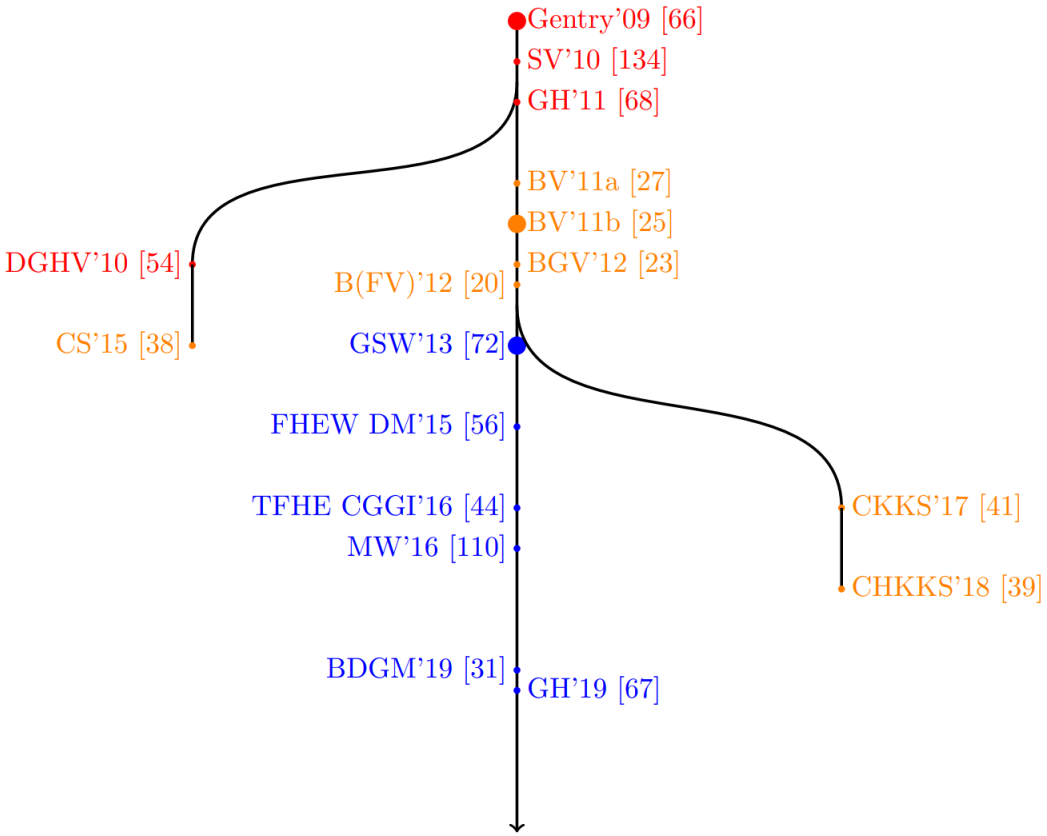

Figure 2: First, second, and third generation FHE schemes. Please refer to [52, p. 27] for the references

Conclusion Each generation and each scheme has its own advantages and disadvantages. The first generation was more of a theoretical experiment, but the following ones have candidates that have already been proven to work in real applications, for example:

- The password manager of Microsoft's web browser Edge uses HE to find out if a password stored by the user can be found in a database of breached passwords.[16]
- Using the lattigo library [51, p. 5]:
  - Multinomial logistic regression for secure genotype imputation
  - Machine learning models:
    * Generalized linear models on distributed datasets [24]
    * Feed-forward neural networks [61]

2.4 Parameters

Before you can use an FHE scheme you have to set several parameters. For each application these parameters can be different, because they influence not only the security and performance of the resulting application, but also its potential functionality. On an abstract level, like the interface of a library, it is enough to set a handful of parameters. On the theoretical level, however, there are dozens of parameters that must be taken into account.

Choosing the right parameters is an extremely complex procedure that requires a deep understanding of the implementation and underlying theory of the scheme used (see for example Table 5 on page 28, Eq. (7) on page 25, Eq. (8) on page 26). Users without a broad understanding not only of cryptography, but of homomorphic encryption in particular, have no chance of choosing proper parameters. Although there are predefined default parameters, they only cover simple scenarios. Sophisticated use of FHE schemes demands carefully selected parameters – otherwise the homomorphic evaluations or the subsequent decryption may not work. In addition, due to a lack of standardization, many publications name the parameters differently.

This chapter captures two points of view: On the one hand a high-level parameter selection using available application programming interfaces (APIs), shown in Section 2.4.1 on the following page. On the other hand a low-level parameter selection, describing the implementation details of FHE schemes in Section 2.4.2 on page 24. The

[16] https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/

former only includes setting very few parameters, compared to the actual implementation that needs dozens of values to be set. This is possible by carefully selecting default values for some parameters that cannot be changed through the API afterwards.

These difficulties are demonstrated using the BFV scheme, of which a high-level API is offered by Microsoft's Simple Encrypted Arithmetic Library (SEAL). This already skips the difficult step of choosing an adequate scheme that is capable of the desired functionality. The previous Section 2.3 on page 18 described the differences between the schemes, concerning the computational models, like modular or floating-point arithmetic.

The BFV scheme is capable of modulo integer arithmetic, and also so-called *packing* of plaintexts. This means that many plaintexts, each consisting of an integer, can be packed into one large plaintext array, which is then encrypted into a ciphertext. This data structure allows for computations in a single instruction, multiple data (SIMD) fashion.

2.4.1 The high-level point of view

From a high-level point of view, when choosing parameters, a trade-off must always be made between the following three points: *security*, *performance* and *functionality*. The relationship between them is as follows:

1. **Security**: Higher security levels result in, e.g., larger key and ciphertext sizes. This directly impacts the performance in a negative way, i.e., homomorphic operations take more time.

2. **Performance**: Higher performance needs consequently require lower security levels or less functionality.

3. **Functionality**: Higher functionality needs, i.e., the desired complexity of homomorphic functions to be evaluated, result in poorer performance (if the security level stays the same), or in a poorer security level (if the performance level stays the same). Different functionalities are for example multiplication, potentiation, branching and looping. Multiplicative depth is also relevant.

Note on the functionality trade-off Of course, there is no functionality trade-off when using FHE, because any function can be computed inherently by the scheme

(this is what it is all about). But especially second generation schemes are often used in a different way: Instead of bootstrapping the ciphertexts regularly, which is a very time- and performance-consuming procedure, the scheme is „simply“ set up in such a way, that it is capable of computing the desired functions without bootstrapping, i.e., in a SHE manner. This is possible due to the advancements made in contrast to first generation schemes in terms of performance and functional capability even with small parameters. In practice, this is done by controlling the so called *noise budget* via the parameter selection, which is explained below. The larger the noise budget is at the beginning of encryption, the more computations can be made on the ciphertexts without bootstrapping.

In SEAL there are four parameters to be set by the user, when using the BFV scheme. These parameters are listed in Table 4 on the next page, including example values. Changing them influences the above mentioned high-level parameters security, performance and functionality. What follows is a superficial description of these parameters, taken from the documentation of an example from SEAL. [49] More details on these and other parameters are given in Table 5 on page 28.

Here are the four parameters in SEAL which are expected to be set by the user:

1. **`security_level`**: This parameter controls the security of the scheme (how exactly is explained in the low-level description of the parameters below). Its explicit assignment can be omitted and is then set to a default value of 128. It affects the performance of the scheme, as other parameters are limited by this value.

2. **`poly_modulus_degree`**: This parameter must be an integer and a positive power of 2. „Larger `poly_modulus_degree` makes ciphertext sizes larger and all operations slower, but enables more complicated encrypted computations“. [49] (ll. 50-53)

3. **`coeff_modulus`**: „This parameter is a large integer, which is a product of distinct prime numbers, each up to 60 bits in size. [...] A larger `coeff_modulus` implies a larger noise budget, hence more encrypted computation capabilities. However, an upper bound for the total bit length of the `coeff_modulus` is determined by the `poly_modulus_degree`.“ [49] (ll. 69-71) This parameter can be automatically set with a helper function, which depends on `security_level` and `poly_modulus_degree`.

4. **`plain_modulus`**: This parameter „determines the size of the plaintext data type and the consumption of noise budget in multiplications. Thus, it is essential to try to keep the plaintext data type as small as possible for best performance.“ [49] (ll. 107-109) It can be any positive integer.

Note on some parameters Table 4 denotes the notation of some parameters logarithmically, e.g., $\log_2 t$ for the `plain_modulus`. This indicates, that in SEAL it sometimes suffices to enter the bit size the parameter will have, e.g., 20. The library then calculates a 20-bit number in the background that matches the remaining parameters.[17]

Notation	Name	Data type	Example value
λ	`security_level`	integer	128
d or n	`poly_modulus_degree`	integer	4096
$\log_2 q$	`coeff_modulus`	array of Integers	$[36, 36, 37]$
$\log_2 t$	`plain_modulus`	integer	20

Table 4: High-level parameters for the BFV scheme in SEAL

2.4.2 The low-level point of view

The security level is used as an example to illustrate the complex relationships between the different parameters. The security level parameter is called λ and typically takes integer values such as 128, 192 or 256.

1: Controlling λ The BFV scheme can be shown to be *IND-CPA secure*. In short, this implies that an attacker who has only the ciphertext and the public key cannot learn anything about the underlying plaintext. [55, p. 131]

The security of the BFV scheme is based on the hardness of the ring learning with errors (RLWE) problem.[18] The security level λ controls the allowed time for an attacker

[17] To enable *batching* in BFV (i.e., the above mentioned *packing* of integers into an array), a prime number is needed as `plain_modulus`. Thus, the automatic computation of a 20 bit prime comes in very handy.

[18] Details on the RLWE assumption can be found in [48]

to break an encryption scheme. More precisely a security level of λ means that all attacks requiring no more than 2^λ operations are unsuccessful.[19] [12, p. 127]

The authors of [23] give the following formula for predicting the runtime in seconds that will break the BFV scheme:

$$\log_2(time) = 1.8/\log_2(\delta) - 110. \tag{6}$$

„If we assume a security level of λ bits, i.e., we set $time = 2^\lambda$, then the minimal δ we can achieve according to the above estimate is $\log_2(\delta) = 1.8/(\lambda + 110)$. For instance, when we set $\lambda = 128$, we obtain that $\delta \simeq 1.0052$." [23, p. 16] Immediately, the reader is confronted with a new, unknown, low-level parameter: δ. In this specific context, δ is defined by the authors as follows:

„Gamma and Nguyen [6] defined the Hermite factor δ^m of a basis $\mathbf{B}$ of an m-dimensional lattice Λ as $\|\mathbf{b}_1\| = \delta^m \cdot \det(\Lambda)^{1/m}$, with $\mathbf{b}_1$ the shortest vector in $\mathbf{B}$." [23, p. 15]

Lattice theory is not covered in this thesis and therefore the reader is referred to [23] for more details. However, to understand what the Hermite factor δ^m is, the reader needs to know lattices.

2: Some parameters depending on δ Nearly all of the parameters involved in this scheme have some relationship to each other. Some of them are expressed in inequalities. The following inequality, for example, describes the condition to choose the parameters α, q and σ in dependence of d and δ, where δ controls the security level, as shown in Eq. (6) [23, p. 16]:

$$\alpha \cdot \frac{q}{\sigma} < 2^{\left(2 \cdot \sqrt{d \log_2(q) \log_2(\delta)}\right)} \tag{7}$$

The parameters that appear in this inequality are:

1. α: This is a constant. The scheme needs the condition Eq. (7) to be secure.

2. q: The modulus of the ciphertext space, which is chosen (together with σ) based on this inequality.

[19] For example, a security level of $\lambda = 128$ means that there can be no successful attacks as long as attackers can perform at most 2^λ operations. [12, p. 127]

3. σ: The standard deviation of the noise, which is chosen (together with q) based on this inequality.

4. d: The degree of a polynomial (corresponds to `poly_modulus_degree`), which is chosen based on this inequality.

5. δ: The root-Hermite factor, which influences the security level.

3: Parameters for FHE Another example is the following equation: [23, p. 16]

$$4 \cdot \alpha \cdot \beta(\varepsilon) \cdot \delta_R^{L_{\min}} \cdot (\delta_R + 1.25)^{L_{\min}+1} \cdot t^{L_{\min}-1} < 2^{\left(2 \cdot \sqrt{d \log_2(q) \log_2(\delta)}\right)} \tag{8}$$

„The [...] formula thus allows us to either choose d first and then compute a valid (q, σ)-pair or vice-versa.“ [23, p. 17]

The authors also provide a *simple example* of how to set parameters for the scheme: „To provide a simple example, consider the family $f_d(x) = x^d + 1$, then $\delta_R = d$, $H(f_d) = 1$ and for $t = 2$, $h = 63$ we have $L_{min} = 9$. For $\epsilon = 2^{-64}$ we have $\beta(\epsilon) \simeq 9.2$ and $\alpha \simeq 3.8$ and for 128-bit security level we have $\log_2(\delta) = 0.0076$. If we choose $q = 2^n$ and $d = 2^k$, then substituting all these values then finally leads to

$$15.13 + 19 \cdot k < 0.174 \cdot \sqrt{n} \cdot 2^{k/2}.$$

So if we choose $k = 10$, then we require $n > 1358$ to guarantee FHE capabilities.“ [23, p. 17]

It becomes clear, that it is almost impossible to give a *brief* overview of some parameters, because they all depend on each other and none of them is unimportant. Even if libraries like SEAL take the part of choosing some parameters, it still remains a difficult task for a non-expert to even set the 2-4 high-level parameters from Table 4 on page 24.

2.4.3 Tools to set the right parameters

The online sandbox tool *morfix* [35] offers a front end of SEAL.[20] It allows the user to experiment with HE without having to know how to program, or how to set encryp-

[20] Actually, the back end of morfix is a JavaScript (JS) library called node-seal [5], which is a web implementation of SEAL.

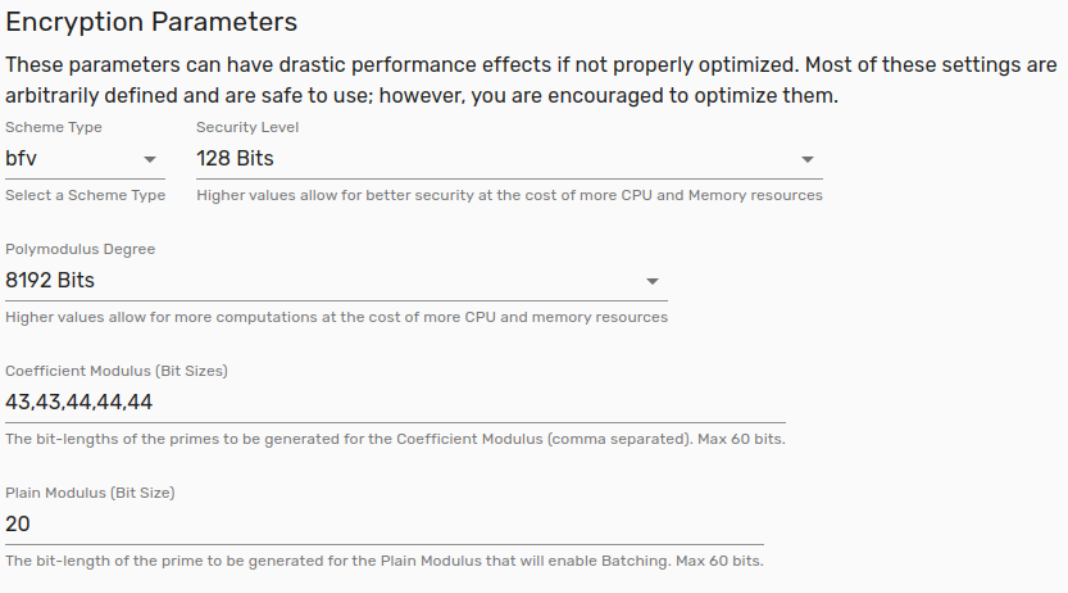

Figure 3: `https://s0l0ist.github.io/seal-sandbox/` automatically computes parameters based on user inputs

tion parameters (if the standard set of parameters is used). A fresh load of the web page will bring up the following setup: Using the HE scheme BFV, the security level is set to 128. The default „Polymodulus Degree“ is set to 4096, and the resulting „Coefficient Modulus (Bit Sizes)“ is [36, 36, 37]. The „Plain Modulus Size (Bit Size)“ is set to 20. Changing the parameters, results in automatically computed values for the other parameters, see the screenshot in Fig. 3. However, these calculations are not explained on the web page. In Fig. 3 the „Polymodulus Degree“ was manually set to 8192 Bits. The „Coefficient Modulus“ [43, 43, 44, 44, 44] was computed by the web page.

A blog post on `https://medium.com` [60] explains very well how to set the parameters in SEAL for the CKKS scheme.

Action	Notation	Name	Context	Explanations/Remarks	Special case	Example
compute	α	a constant	$\alpha = \sqrt{\frac{\ln(\frac{1}{\varepsilon})}{\pi}}$	$\alpha \cdot \frac{q}{\sigma} < 2^{\left(2\cdot\sqrt{d\log_2(q)\log_2(\delta)}\right)}$ (see [23, Eq. (6) p. 16])		α =3.8
	B	bound	$B = \beta(\varepsilon)\cdot\sigma$	a distribution χ over the integers is called B-bounded if it is supported on $[-B, B]$ (see [23, p. 3])		$B = 10\,\sigma$
compute	$\beta(\varepsilon)$	a function	$\beta(\varepsilon) = \min\{\beta \in \mathbb{R} : \operatorname{erf}\left(\frac{\beta}{\sqrt{2}}\right) < \varepsilon\}$	with probability $1-\varepsilon$ samples are bounded by $\beta\cdot\sigma$		$\beta(\varepsilon) = 9.2$
	d	degree of polynomial	usually $f(x) = x^d + 1, d = 2^k$	notation not consistent in literature, sometimes n instead of d	$d = n$ a power of 2 ([8, p. 2]) or $d = 2^k$ ([23])	$d = 2^{10}$
compute	$\log_2\delta$	δ: root-Hermite factor				$\delta = 1.0052$
	δ_R	expansion factor of R, ring constant	$\delta = \max\left\{\frac{\lvert a\cdot b\rvert}{\lvert a\rvert\cdot\lvert b\rvert} \;:\; a, b \in R\right\}$	infinity norm $\lVert a\rVert = \max\{\lvert a_i\rvert : i = 1,\dots,d-1\}, a = \sum_{i=0}^{d-1} a_i x^i \in R$	$\delta_R = d$	
	Δ		$\Delta = \lfloor q/t \rfloor$	the ratio Δ will basically determine the maximal number of homomorphic operations which can be done in a row to ensure a correct decryption ([8, p. 3])		
choose	ε	upper bound for probability	$\beta(\varepsilon) = \min\{\beta \in \mathbb{R} : \operatorname{erf}\left(\frac{\beta}{\sqrt{2}}\right) < \varepsilon\}$	$\operatorname{erf}\left(\frac{\beta}{\sqrt{2}}\right)$ is the probability that a sample originates from outside the range of width $2\beta\sigma$		$\varepsilon = 2^{-64}$
	f	polynomial (`poly_modulus`)			$f(x) = x^d + 1$	
choose	h	Hamming weight		large enough s.t. χ has sufficient entropy		$h = 63$
	k	$k \in \mathbb{R}, k > 1$ or $k \in \mathbb{N}$	transform valid pair (q, σ) into another valid pair (q^k, σ_k) (see [23, p. 16]) or $d = 2^k$	usage not consistent in literature, sometimes number of bits of d, sometimes number of small moduli $q_1, \dots, q_k$		$k = 10$
choose	λ	security level	$\log_2\delta = \frac{1.8}{\lambda+110}$	„time“$= 2^\lambda$, λ proportional to number of bits of time needed for successful attack (see [23, p. 15 f.])		$\lambda = 128$
	L $L_{\min}, L_{\max}$	multiplicative depth				$L_{\min} = 9$
	n	number of bits of q	$q = 2^n$	$4\cdot\alpha\cdot\beta(\varepsilon)\cdot\delta_R^{L_{\min}}\cdot(\delta_R + 1.25)^{L_{\min}+1}\cdot t^{L_{\min}-1} < 2^{\left(2\cdot\sqrt{d\log_2(q)\log_2(\delta)}\right)}$ (see [23, p. 16])		$n > 1358$
	q	modulus in the ciphertext space (`coeff_modulus`)			$q = 2^n$ ([23])	$q = 2^{1358+\dots}$
	$q_1, q_2, \dots, q_k$	small moduli, RNS base (see [8, p. 4])	$\prod_{i=1}^{k} q_i = q$	used for Chinese Reminder Theorem in full RNS variant of BFV		
	R	ring	$R = \mathbb{Z}[x]/(f(x))$	ring of polynomials with coefficients in $\mathbb{Z}$ and degree $\leq \deg(f) - 1$		
	$R_t, R_q,$ $R_q[Y]$	rings	$R_t = \mathbb{Z}_t[x]/(f(x))$ or $R_q = \mathbb{Z}_p[x]/(f(x))$,	ring of polynomials with coefficients in $\mathbb{Z}_p$ or $\mathbb{Z}_t$ and degree $\leq \deg(f) - 1$; R_t plaintext space; $R_q[Y]$ ciphertext space (see [8, p. 3])		
	σ	noise standard deviation	$\sigma = \sqrt{\operatorname{VAR}(\chi)}$			
	t	modulus in plaintext space (`plain_modulus`)	$t << q$		$t = 2$ or $t \mid q$	
choose	χ	probability distribution on R, noise is sampled from χ	in general not as simple as just sampling according to $D_{\mathbb{Z},\sigma}$ (discrete Gaussian distribution)	special case $f(x) = x^d + 1$ with d a power of 2: $\chi \approx D_{\mathbb{Z},\sigma}$ with $D_{\mathbb{Z},\sigma} : \left\{ \begin{array}{ccc} \mathbb{Z} & \to & [0;1] \\ x & \mapsto & \sim e^{-\pi\frac{x^2}{\sigma^2}} \end{array} \right\}$		

Table 5: Parameters that occur in the theoretical description of the BFV scheme

2.5 Primitives from the homomorphic encryption standard

In this section, some primitives inherent to all FHE and SHE schemes are presented. These primitives are based on a standard that was published by *HomomorphicEncryption.org*. [4] This is a group of researchers from industry (e.g., Microsoft, Samsung, Intel), government (e.g., National Institute of Standards and Technology (NIST)) and academia (e.g., Boston University, University of Hannover) who have addressed the task of standardizing FHE. That HE needs standardization is evident from the previous Section 2.4 on page 21, which describes the chaos concerning the parameters in HE schemes. The consortium organizes annual workshops and standards meetings, where new developments are presented and discussed. The first standard was published in late 2018 and includes, e.g., general notations and definitions and a description of the FHE schemes BGV, BFV and GSW.

At the time of writing, there does not seem to be a newer standard. But of course there are a lot of discussions going on, see for example `https://www.iso.org/standard/83139.html`.

2.5.1 Public-key encryption algorithms

When FHE and SHE schemes are based on a PKE scheme, they require the following generic algorithms: `keygen`, `enc`, and `dec`.

These algorithms do not yet have homomorphic properties.

The key generation algorithm $\texttt{KeyGen}(\texttt{params}) \rightarrow \texttt{sk}, \texttt{pk}, \texttt{ek}$	
The ***input*** of the algorithm	The ***outputs*** of the algorithm
• `params`: an object that holds the parameters of the scheme	• `sk`: the secret key • `pk`: the public key • `ek`: the evaluation key

Table 6: KeyGen

A difference in comparison to traditional PKE schemes can be observed: Beside the public key `pk` and the secret key `sk`, the key generation algorithm in Table 6 also outputs a so-called evaluation key `ek`. This key is needed to perform homomorphic evaluations on ciphertexts, i.e., calculating the result of a function homomorphically. Details on this and how the `params` object is created with the `ParamGen` algorithm are described below in Section 2.5.2 on the next page.

The encryption algorithm $\texttt{Encrypt}(\texttt{pk}, \texttt{m}) \rightarrow \texttt{c}$	
The ***inputs*** of the algorithm	The ***output*** of the algorithm
• `pk`: the public key • `m`: the message	• `c`: an encryption of `m` under `pk`

Table 7: Encrypt

The decryption algorithm $\texttt{Decrypt}(\texttt{sk}, \texttt{c}) \rightarrow \texttt{m}$	
The ***inputs*** of the algorithm	The ***output*** of the algorithm
• sk: the secret key • c: the ciphertext	• m: the decryption of c

Table 8: Decrypt

The encryption and decryption algorithms in Table 7 on the preceding page and Table 8 are known from common PKE schemes and also behave as such. However, the implementations of these algorithms vary between FHE schemes, as some of the schemes are based on different mathematical concepts, e.g., ideal lattices and polynomial rings. Details on one of these implementations (BFV) can be found in Section 2.6 on page 36.

2.5.2 Homomorphic encryption algorithms

HE schemes come along with a set of algorithms, required for homomorphic evaluations. The first algorithm that is needed is the parameter generation algorithm (Table 9 on the next page). Unfortunately, some starting parameters still have to be set manually, unlike to what the name of the algorithm suggests. The difficulties of this task were explained in Section 2.4 on page 21. The object output by the algorithm encapsulates the entered parameters. This way, they can easily be forwarded to other algorithms mentioned below, that all need these parameters.

The parameter generation algorithm $\texttt{ParamGen}(\lambda, \texttt{PT}, \texttt{K}, \texttt{B}) \rightarrow \texttt{params}$	
The ***inputs*** of the algorithm	The ***output*** of the algorithm
• $\lambda \in \{128, 192, 256\}$: the security level • $\texttt{PT} \in \{\texttt{MI}, \texttt{EX}, \texttt{AN}\}$: the plaintext space • K: the dimension of a plaintext vector • B: an auxiliary parameter used to control the complexity of the programs or circuits that one can expect to run over encrypted messages	• params: an object that holds the parameters of the scheme

Table 9: ParamGen

The inputs of the ParamGen algorithm were already mentioned on a low level in the previous Section 2.4 on page 21. A new aspect is the choice between different plaintext types PT. These types determine the data structure in which a plaintext will be stored and the data type of plaintexts, e.g., bits, integers or floating-point numbers. The standard defines the following three plaintext types:

- MI: stands for modular integers and allows plaintexts to be integers modulo some prime p. All operations are also performed modulo p.
- EX: stands for extension fields/rings. Simply put, plaintexts are encoded as an integer polynomial with coefficients from the range $(0, p-1)$ for some prime p.

- AN[21]: stands for approximate numbers and allows plaintexts to be floating-point numbers. Homomorphic evaluations can be performed up to a predetermined precision, depending on other parameters.

The dimension of a plaintext vector containing integers (MI) or extension fields/rings (EX) is parameterized with the input K. This allows the aforementioned packing of plaintexts into a large array.

The parameter B controls how complex the evaluations on the ciphertext can be. This also has an effect on the size of keys and ciphertexts. Small values yield smaller key sizes, smaller ciphertext sizes but also less complex programs or circuits that can be expected to run over encrypted messages. [4, p. 3] Larger values result in larger key and ciphertext sizes but the potential complexity of homomorphic evaluations is greater.

What follows are algorithms that allow the user to evaluate functions over ciphertexts.

The homomorphic addition algorithm $\texttt{EvalAdd}(\texttt{params}, \texttt{ek}, \texttt{c}_1, \texttt{c}_2) \rightarrow \texttt{c}_3$	
The ***inputs*** of the algorithm	The ***output*** of the algorithm
• params: an object that holds the parameters of the scheme • ek: the evaluation key • $\texttt{c}_1$: an encryption of a plaintext $\texttt{m}_1$ • $\texttt{c}_2$: an encryption of a plaintext $\texttt{m}_2$	• $\texttt{c}_3$: an encryption of a plaintext $\texttt{m}_3 = \texttt{m}_1 + \texttt{m}_2$

Table 10: EvalAdd

[21] This plaintext space type is not part of the standard published in 2018. But it is mentioned there as part of future versions of the standard. The type can be related to the CKKS scheme that operates on floating-point numbers as plaintexts.

The homomorphic multiplication algorithm $\texttt{EvalMult}(\texttt{params}, \texttt{ek}, \texttt{c}_1, \texttt{c}_2) \rightarrow \texttt{c}_3$	
The ***inputs*** of the algorithm	The ***output*** of the algorithm
• params: an object that holds the parameters of the scheme • ek: the evaluation key • $\texttt{c}_1$: an encryption of a plaintext $\texttt{m}_1$ • $\texttt{c}_2$: an encryption of a plaintext $\texttt{m}_2$	• $\texttt{c}_3$: an encryption of a plaintext $\texttt{m}_3 = \texttt{m}_1 \cdot \texttt{m}_2$

Table 11: EvalMult

The above algorithms `EvalAdd` (Table 10 on the preceding page) and `EvalMult` (Table 11) let the user homomorphically add (or multiply) two encrypted values. The result is the encrypted sum (or product) of the underlying plaintexts. As an example, Section 2.1.1 on page 9 (RSA example) represents a homomorphic multiplication. Here, the evaluation key `ek` is required to perform the homomorphic operations. „It should be given to any entity that will perform homomorphic operations over the ciphertexts. Any entity that has only the public and the evaluation keys cannot learn anything about the messages from the ciphertexts only.“ [4, p. 4]

In addition, there are the algorithms `EvalAddConst` and `EvalMultConst`. They simply provide the addition of a plaintext constant to a ciphertext or the multiplication by plaintext constant with a ciphertext. This way, it is not necessary to have the second parameter of the calculation encrypted.

The algorithm in Table 12 on the next page is also very important for FHE, as it refreshes a given ciphertext to reduce its complexity. This complexity refers to the inherent noise attached to every ciphertext. Once it gets too large, no further homomorphic operations can be made, and in the worst case, correct decryption may not be possi-

ble anymore. Therefore, it is crucial to lower this noise regularly with the `Refresh` algorithm.

The refresh algorithm $\texttt{Refresh}(\texttt{params}, \texttt{ek}, \texttt{c}_1) \rightarrow \texttt{c}_2$	
The ***inputs*** of the algorithm	The ***output*** of the algorithm
• `params`: an object that holds the parameters of the scheme • `ek`: the evaluation key • $\texttt{c}_1$: an encryption of a plaintext $\texttt{m}_1$	• $\texttt{c}_2$: another encryption of a plaintext $\texttt{m}_1$

Table 12: Refresh

Especially this algorithm, `Refresh`, is realized in many different and complex ways. During the first generation of FHE, regarding Gentry's initial scheme, the refresh algorithm corresponds to the bootstrapping procedure that resets the noise of a ciphertext by evaluating the decryption circuit homomorphically. This changed in second generation schemes, like BFV, where a technique called *relinearization* was introduced.

Relinearization in BFV

On a high level, what essentially happens during relinearization is the following: In the BFV scheme, ciphertexts not only have a noise but also an integer valued *size*. The size corresponds to the number of polynomials that represent the ciphertext. A new, i.e., freshly encrypted ciphertext has size 2, and homomorphic multiplications let this size grow. Specifically, „if the input ciphertexts have sizes M and N, then the output ciphertext after homomorphic multiplication will have size $M+N-1$." [49] (ll. 279-280) The relinearization procedure reduces the size of a ciphertext from 3 back to 2.

The reason why this is necessary is, that the greater the size of ciphertext is, the faster the noise parameter grows. This means, that multiplying two ciphertexts, one or both of which has a size greater than 2, adds more noise to the resulting ciphertext than multiplying ciphertexts of small size, where the best case is a size of 2.

The authors who proposed the BFV scheme [23], provide two different approaches for the relinearization algorithm, both of which require the introduction of a relinearization key: `rlk`. This key can be generated alongside the other keys `pk`, `sk` and `ek` in the `KeyGen` algorithm. The first variant is called the *dynamic relinearization* and the second variant works with so-called *modulus-switching*. The details to this techniques are beyond the scope of this thesis. The reader is referred to [23, pp. 8-9].

Multi-party homomorphic encryption

Beyond the two party approach of FHE, which includes a client that encrypts their data and a server that homomorphically evaluates functions on that data and sends them back to the client, a multi-party setting is also possible to think of. In fact, the lattigo library [51] provides implementations of the schemes BFV and CKKS in the programming language Go, that support multi-party functionalities for both schemes.

A multi-party HE library extends the standard two party setting to N parties. „In such schemes, the involved parties collectively (hence, interactively) enforce the access control to the data by distributing the scheme's decryption circuit.“ [51, p.1] Techniques that provide these functionalities have been proposed in [13, 47, 6, 50].

However, in this thesis, this approach will not be necessary, as the two party setting suffices for the implementation in Section 3 on page 38.

2.6 The RNS BFV scheme

In Section 3 on page 38 an implementation of an HE application will be presented. This implementation will make use of the residue number system (RNS) BFV scheme with the help of the library node-seal. This feature (RNS), which speeds up computations in the scheme, is described below.

The concrete computations underlying the BFV scheme involve multiplications of very large numbers, due to the large coefficients of the polynomials. A work by Bajard et al.

[8] introduces the usage of the chinese remainder theorem (CRT) to speed up the multiplication of these large numbers. Representing numbers via the CRT is also known as the RNS representation. Hence, the alternative implementation of the BFV scheme using the CRT to gain performance advantages is called the full RNS variant of BFV, if every multiplication is done using the RNS representation. It is also implemented in SEAL.

The full RNS variant of BFV by [8] uses the first variant of the above mentioned relinearization techniques (see Section 2.5.2 on page 35) from the BFV scheme. [8, p. 9]

2.7 Security of FHE

To conclude this theoretical chapter, we will briefly address the security of FHE. In 2009, Craig Gentry already had the following to offer on this subject:

- „Unfortunately, a scheme that has nontrivial homomorphisms cannot be CCA2 secure, because it is malleable“ [26, p. 32]
- „However, [...] finding a CCA1-secure fully homomorphic encryption scheme [is] an interesting open problem“ [26, p. 33]

A lot has happened since then. For the current security issues around FHE, we refer to Computing Compass, where one can find the following statement, for example:

- „CCA security is not possible for FHE. Due to the homomorphism, chosen-ciphertext attacks are possible“ [52, p. 26]
  Whether FHE is IND-CPA safe is not stated.

In addition, in the context of security, circuit privacy/evaluation privacy must also be mentioned:

Circuit privacy [26, p. 32] (also called evaluation privacy [4, p. 15]) ensures that a freshly encrypted ciphertext cannot be distinguished from a ciphertext which is the result of homomorphic operations.

3 Implementation

This chapter describes the implementation of a use case for FHE. Section 3.1 explains which application is implemented and why it was chosen. Section 3.2 on page 44 and Section 3.3 on page 46 explain which library was chosen and why. Then the implemented application is presented in Section 3.4 on page 49 with the help of screenshots. Finally, in Section 3.5 on page 54 the integration of the application into the CTO website is described.

3.1 Application selection

The goal is to create a web application for the website CrypTool-Online (CTO), a website with applications for testing and learning classic and modern cryptography.[22] The application is intended to introduce the user to HE with a step-by-step program, that presents different properties and behaviors of an FHE scheme.

At first, the idea was to build a web application that represents a complete front end of a selected FHE library. The user would have been able to generate any number of plaintexts and ciphertexts and connect them with any homomorphic function the library offered. But it turns out that this task would be very complex and too extensive for this thesis.[23]

As the parameters are too complex (see Section 2.4 on page 21) most users would not understand such an app. So we decided to choose an app where HE is integrated from the very beginning to achieve security by design.

The Lattigo polls demo The choice then fell on a Doodle-like application that only covers a simple use case of HE. The final inspiration was found in the paper introducing the Lattigo library for multi-party homomorphic encryption (MHE). [51] The paper presents a demo application written in Go that demonstrates the use of the Lattigo library. The selected use case is an application for „privacy preserving scheduling“ [51, p. 5], i.e., a tool to conduct a poll online that hides the inputs, calculations and results using HE. The code for this Go application called *Lattigo-polls-demo* can be found on GitHub [38] and is briefly described in Section 3.3 on page 46.

[22] https://www.cryptool.org/de/cto/

[23] For the three schemes RSA, Paillier, and Gentry-Halevi such an application idea is implemented in JavaCrypTool.

In general, a poll is a tool that can be used to identify a subset of a set of choices that is suitable for each participant taking part in the poll. A simple example is the task of finding the days in a given period of time, e.g., a week from Monday to Friday, on which each participant is free. First, a poll is created by the poll creator, who defines the set of options that a participant can choose from. Then, an arbitrary number of people individually submit their answers to the poll. When everyone has submitted their answer, the result of the poll is determined by calculating the intersection of all the answers given.

Privacy of free polling tools Such tools can be found free on the internet. Examples are Doodle[24], StrawPoll[25], and Nuudel.[26] In these web applications, the server calculates the result after all participants in the poll have submitted their input, i.e., they send their unencrypted data to the server. This represents a lack of data privacy because a malicious server could read the user's input. The principle of privacy by design is thus not satisfied.

The Hypertext Transfer Protocol Secure (HTTPS) protocol does not help here either, because with its help the data is only *transmitted* securely, i.e., encrypted, but unencrypted at the endpoint when received. Neither the submissions nor the results of the poll are encrypted outside of the transmission of the data, because the encryption must be removed again by the server for processing the data. It is usually possible to control who can see the results after the poll has closed, either all participants or only the poll creator. But this is done through access control using passwords rather than encryption.

The solution using homomorphic encryption The privacy problems described above can be solved using HE. The aim of this section is to explain how to implement a polling tool that preserves data privacy. Using HE to conduct a poll has the following advantages:

1. The user encrypts their data before sending it to the server. HTTPS is not even required.

2. The server receives the encrypted inputs and homomorphically evaluates the result without ever seeing the plaintext data from any participant.

[24] https://doodle.com/en/
[25] https://strawpoll.com/en/
[26] https://nuudel.digitalcourage.de: Non-tracking tool of the non-profit Digitalcourage e.V.

3. The result itself is also encrypted and,

    a) in our setting, can only be decrypted by the poll creator.

    b) in an MHE setting, can only be decrypted by all participants.

4. The input of a participant is not shown on the website and cannot be read – neither by another participant nor by the poll creator.

The MHE variant of the implementation, where all participants can decrypt the result, is omitted: This technique would require the introduction of additional MPC protocols and the use of a dedicated MHE library, which would also need to be explained. Again, this is beyond the scope of this thesis. Therefore, the focus will be on the situation where only the poll creator has the ability to decrypt the poll result (and may show the result on his website).

The procedure on a high level To conduct an encrypted poll, an HE scheme can be used. Contrary to what the title of the inspiring paper might suggest („*Lattigo: a Multiparty Homomorphic Encryption Library in Go*“ [51]), there is no need for a multiparty setting for such an application. This becomes clear considering the following application scenario from [52, p. 52].

Private data aggregation Let there be *n clients*, an *aggregation server* and a *processing server*. Suppose that the clients want to aggregate their data without revealing it to the aggregation server. This can be accomplished in the following way: The processing server generates keys for an FHE scheme, i.e., a public key, a private key and an evaluation or relinearization key. The public key is made available, especially to the clients. The evaluation key is made available to the aggregation server. The clients encrypt their inputs using the public key received from the processing server and send their encrypted input to the aggregation server. The aggregation server homomorphically combines the encrypted user data upon receiving them using the evaluation key. Then, the aggregation server sends the aggregated and encrypted data to the processing server. The processing server can process the aggregated data after decrypting it with the secret key.[27]

[27] It is supposed that the post processing of the aggregated data on the processing server is too complex to be evaluated homomorphically, as this can be very resource-intensive, and is therefore done conventionally, i.e., on the decrypted data.

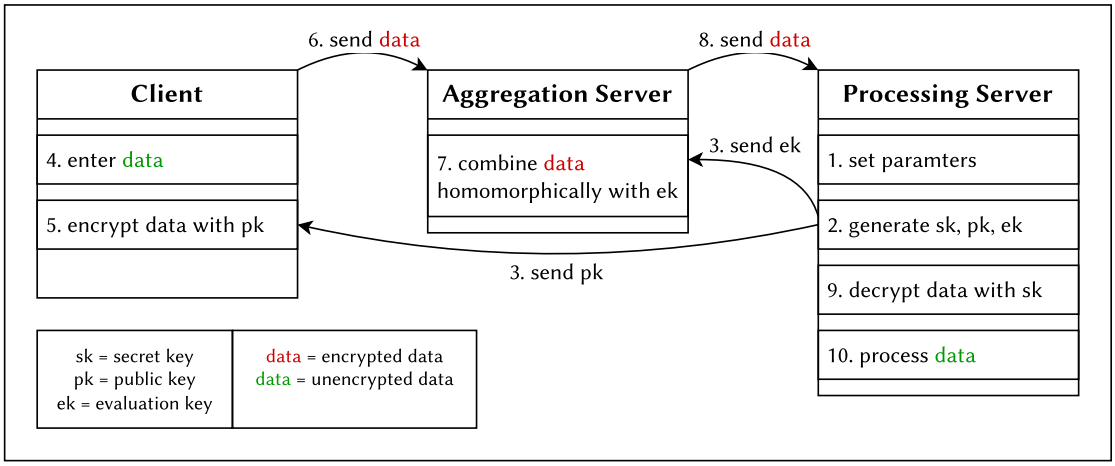

Figure 4: Private data aggregation

The private data aggregation scenario is shown in Fig. 4. Here, the client on the left is representative of all n clients whose data is to be aggregated.

The above scenario can be used in a slightly modified form to realize private polls: n participants want to find out through a poll on which days of the week everyone is free. They want to keep their answers secret from the server that calculates the result of the poll. There is a poll creator who generates the public and private key for an FHE scheme. The public key is made accessible to the participants so that they can use it to encrypt their answers and send them to the server. The server homomorphically computes the result of the poll and sends it to the poll creator. Only the poll creator has the secret key and thus the ability to decrypt answers and results.

This case of conducting a private poll as a special case of private data aggregation is shown in Fig. 5 on the next page. As in Fig. 4, the one participant on the left is representative of all n participants in the poll. Fig. 6 on page 43 shows the data flow between the poll creator, the aggregation server and the participants.

The outline for conducting a privacy-preserving poll is as follows (the numbers in parentheses after each step refer to the steps in Fig. 5 on the following page):

1. **Instantiating the poll**: The *poll creator* ...

   a) ... sets the parameters of the scheme. (1)

   b) ... generates the keys for the HE scheme. (2)

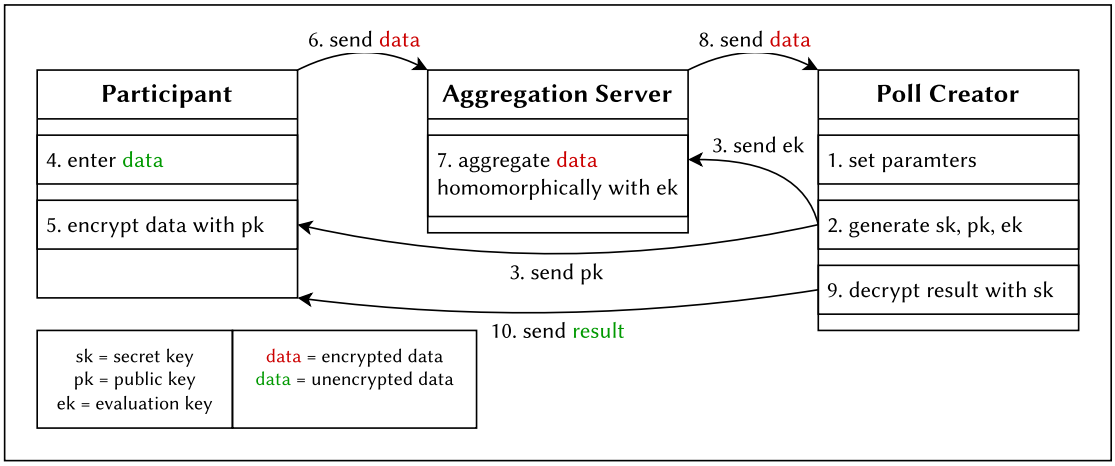

Figure 5: Special case of private data aggregation: polling

c) ... publishes the public key to the participants. (3)

d) ... sends the evaluation key to the server that computes the poll result. (3)

2. **Conducting the poll**: Each *participant* ...

   a) ... enters their answers to the poll. (4)

   b) ... encrypts their data. (5)

   c) ... sends the encrypted data to the server. (6)

3. **Closing the poll** and **calculating the results**. The *server* ...

   a) ... aggregates / combines the received data and thereby computes the encrypted poll result. (7)

   b) ... sends the encrypted poll result to the poll creator. (8)

4. **Sharing the results**. The *poll creator* ...

   a) ... decrypts the poll result with the secret key. (9)

   b) ... may send the result to the participants. (10)

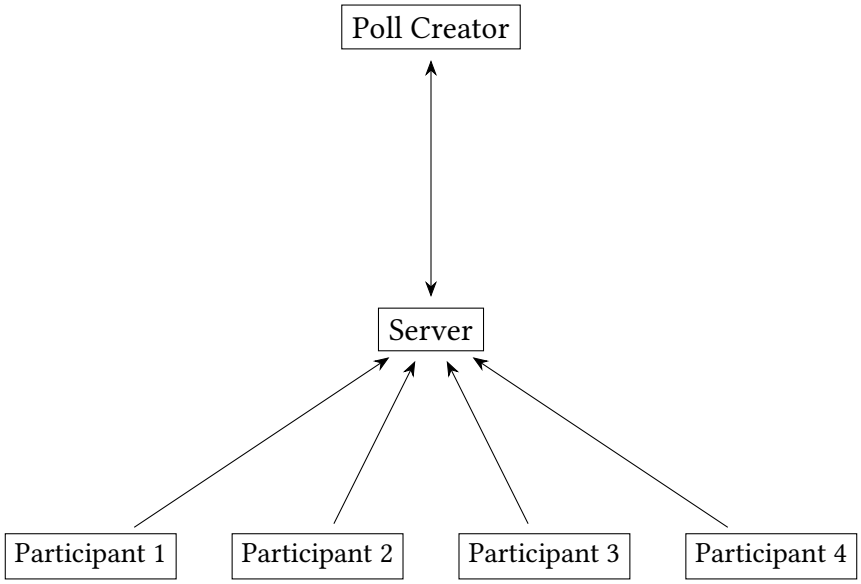

Figure 6: Private data aggregation in our setting of conducting a private poll

In the two scenarios presented, it should be noted that only the aggregation server performs homomorphic calculations. Nothing is computed homomorphically by the other parties.

The procedure on a low level What happens on a lower level when calculating the poll result is briefly described in the following, before a more detailed explanation of the implementation of the application is given in the subsequent Sections 3.2 to 3.5 on pages 44–54.

Submitting an answer Suppose that a participant in the poll is asked to select their free days of the week by clicking on check boxes representing the day of the week. This selection is then converted into a bit array of seven entries, where a 0, i.e., an unchecked check box, indicates that the day is *free* and a 1, i.e., a checked check box, indicates that the day is *occupied*. The indices of the array correspond to the weekdays from Monday (index 0) to Sunday (index 6). The array structure and an example input by a participant is shown in Table 13 on the next page. The example shows an array encoding the participants' input: **Occupied** on Monday, Wednesday, Saturday and Sunday; **free** on Tuesday, Thursday and Friday. This array is then encrypted and sent to the server.

Day of the Week	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Array index	0	1	2	3	4	5	6
Example	[0,	1,	0,	1,	1,	0,	0]

Table 13: Structure of a bit array representing the occupation data of a poll participant for the seven days of a week

Calculating the poll result When all participants have submitted their encrypted answers to the poll, the server can calculate the result homomorphically. To do this, it suffices to multiply all the arrays together, component wise. To understand why this works, consider the following small example: Let the first participant have their free days encoded in the array $P_1 = [0, 1, 1, 1, 1, 0, 0]$ and the second participant in $P_2 = [1, 0, 1, 1, 0, 0, 1]$, respectively. Multiplying these arrays component wise brings the result $[0, 0, 1, 1, 0, 0, 0]$. A more compact overview of this calculation can be observed in Table 14. The crucial property of this calculation is that as soon as one participant's array contains a 0 for a given day, the array containing the poll result must also contain a 0 for that day. This is because $0 \cdot x = 0$, for all x. If, and only if, all participants' arrays contain a 1 for a given day, a 1 will also appear in the result array. This is because $1 \cdot 1 \cdot \ldots \cdot 1 \cdot 1 = 1$.

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
P_1	[0,	1,	1,	1,	1,	0,	0]
P_2	[1,	0,	1,	1,	0,	0,	1]
$P_1 \cdot P_2$	[0,	0,	1,	1,	0,	0,	0]

Table 14: An example of two poll participants and the result that is calculated based on their inputs

In the following sections the implementation details are further described.

3.2 Choosing the appropriate library

This section introduces node-seal. It is a library that ports the SEAL library from `C++` to the Wasm format so it can be used with JS. The Lattigo library (written in Go) is mentioned, which is the library used by the Lattigo-polls-demo.

SEAL for JavaScript As described in Section 3.1 on page 38, the first idea was to implement a complete front end for an HE library. The resulting application is meant to be made publicly available to the CTO website.[28] For this reason, an FHE library was sought, that can be integrated into JS as simply as possible, because this is the programming language the web page is constructed with. Luckily, the developer Nick Angelou ported the SEAL library, which is written in C++, to WebAssembly (Wasm) in a new library called node-seal. [5] Wasm is a low-level byte code that is platform-independent and can be run in modern web browsers. [34] As a compilation target, most C++ programs can be compiled to the Wasm format, with tools like Emscripten. [1] The resulting Wasm module and its functions can then be called in JS using so-called *glue code*, which is automatically generated. For more details on how C/C++ programs can be ported to the web using Wasm and Emscripten, refer to [34, 1].

However, once the decision had been made to use the node-seal library for the application, the implementation aim was changed. Rather than offering a complete front end of, in this case, the SEAL library, a smaller application representing a specific HE use case was chosen. The new goal was to implement a polling app based on the Lattigo-polls-demo, as described in Section 3.1 on page 38. Lattigo and SEAL both contain implementations of the following schemes: BFV, BGV and CKKS. In the Lattigo-polls-demo the BFV scheme is used, which also can be used in the node-seal library.

Lattigo The Lattigo-polls-demo is written in Go and with the MHE library Lattigo [39, 38]. Like C/C++ the Go language natively compiles to Wasm. The developers of the Lattigo-polls-demo thus implemented their application as a web service using Go and Wasm. This involves an additional step of using a tool chain than creates the needed Wasm binaries. For this thesis the decision has been made, to use the available Wasm library node-seal, that already compiled the whole SEAL library to the Wasm format. This avoids writing Go code that implements the HE logic of the application and compiling it to Wasm. Instead, the HE logic can be implemented directly in JS using the node-seal library.

Apart from the above, it is not necessary to use Go to implement a polling application. The high-level and low-level descriptions in the Section 3.1 on page 38 show, that it is not a complex task to implement. Almost every HE library, including SEAL and thus node-seal, supports the multiplication of packed ciphertexts, which is the only homomorphic operation needed for this task.

[28] https://www.cryptool.org/en/cto/

```
logN:  13,
t:     65537,
qi:    []uint64{0x3fffffffef8001, 0x4000000011c001, 0x40000000120001}, // 54 + 54
     + 54 bits
pi:    []uint64{0x7ffffffffb4001}, // 55 bits
sigma: DefaultSigma,
```

Listing 1: Default parameters for the BFV scheme used in the Lattigo-polls-demo (params.go in [39])

For this thesis, a new graphical user interface (GUI) was designed and implemented based on the idea of the Lattigo-polls-demo. It is presented in Section 3.4 on page 49.

3.3 From Lattigo to node-seal

In this section the original Go implementation of the Lattigo-polls-demo is described. Then the realization with the node-seal library of our application is explained.

The implementation of the Lattigo-polls-demo Here, the focus is on the cryptographic aspects rather than on the server and client structure, which is also implemented. When the server side of the application starts, a so called *evaluator* is generated from default parameters. An evaluator is an object in Lattigo that is parameterized with values described in Section 2.4 on page 21. This object can then be used to perform homomorphic evaluations on ciphertexts. In this special case, when using the BFV scheme, these operations can be made without additional keys. The parameters used in the Lattigo-polls-demo can be found in the following listing and in Table 15 on the following page.

As an example of an implementation of the HE logic, the calculation of the poll result is used. How the calculation works in Go can be seen in Listing 2 on page 48. How the same example looks implemented in JS can be seen in Listing 3 on page 48.

The calculation of the poll result The aggregation of the poll responses is performed homomorphically by the server. Since homomorphic calculation requires that

Parameter	Value	Decimal value
$\log N$	13	
t	65537	
q_1	`0x3fffffffef8001`	18014398508400641
q_2	`0x4000000011c001`	18014398510645249
q_3	`0x40000000120001`	18014398510661633
p_1	`0x80000000130001`	36028797020209153
p_2	`0x7ffffffe90001`	36028797017456641

Table 15: Default parameters for the BFV scheme used in the Lattigo-polls-demo (params.go in [39])

as few multiplications as possible take place on a ciphertext, a special method is used for this. This method is explained using the JS code in Listing 3 on the next page.

The `calcResultHE` function receives two arguments. The argument `inputData` is an array whose elements are the encoded answers of the survey participants. The argument `pi`[29] is an object which provides the necessary information to execute the functions of the BFV procedure.

In a while loop, the following calculation is now performed until only one element is left. The first two elements of the array are homomorphically multiplied together (`pi.evaluator.multiply(agg[0], agg[1])`). This product is relinearized immediately to keep the size of the ciphertext small (`pi.evaluator.relinearize(...)`). This ciphertext is then appended to the end of the array (`agg.push(...)`). The two used elements are then deleted from the array (`agg.splice(0, 2)`).

In this way, as few multiplications as possible are performed on the same ciphertext, since they are always moved to the very back of the array. The result of the procedure is finally the product of all elements of the array passed to the function.

[29] `pi` stands for poll instance

```go
// Close computes the polls' result and closes the poll
func (ps *PollServer) Close(p *Poll) {
    p.Closed = true
    if len(p.Participants) > 0 {
        agg := p.responses
        // aggregates the responses recursively
        for len(agg) > 1 {
            agg = append(agg[2:],
                ps.RelinearizeNew(
                ps.MulNew(agg[0], agg[1]), &p.rlk),
                )
        }
        p.result = agg[0]
    }
}
```

Listing 2: Original implementation of the algorithm calculating the poll result in Go (server.go in [38])

```js
export async function calcResultHE(inputData, pi) {
  let agg = [...inputData]
  while (agg.length > 1) {
    agg.push(
      pi.evaluator.relinearize(
        pi.evaluator.multiply(agg[0], agg[1]),
        pi.relinKey
      )
    )
    agg.splice(0, 2)
  }
  return agg[0]
}
```

Listing 3: Efficient multiplication of arrays with minimal multiplicative depth in JS.

3.4 Screenshots of our application

The application is a kind of Doodle enforcing security by design by using the HE library node-seal. The GUI and the API calls to the HE library use own code written in JS.

This section presents the implemented application with some screenshots. The app first shows a user guidance and then explains what information is given to the user so that the benefit of HE in this example becomes clear. At the same time, a small sample poll is conducted.

The introduction page The application is divided into two sub pages, named „Introduction“ and „Polling page“. When the start page is opened, it looks like Fig. 7. The user is provided with a short information that this polling tool uses HE to satisfy the *privacy by design* principle.[30] The user is requested to start the application by clicking the **Start** button on the bottom of the page.

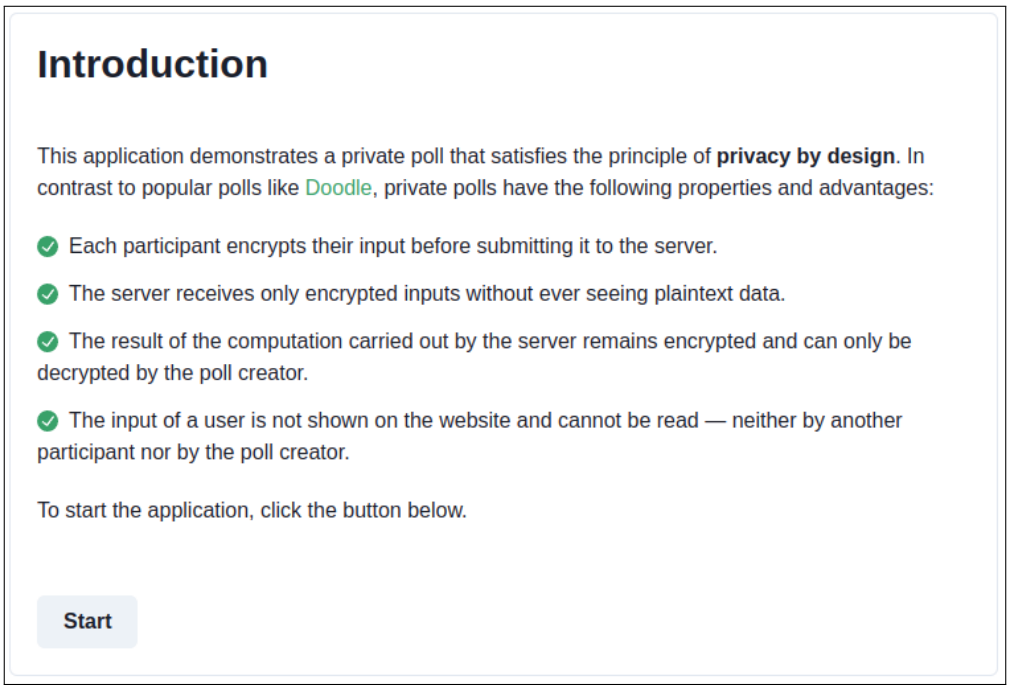

Figure 7: The landing page of the application

[30] https://en.wikipedia.org/wiki/Secure_by_design

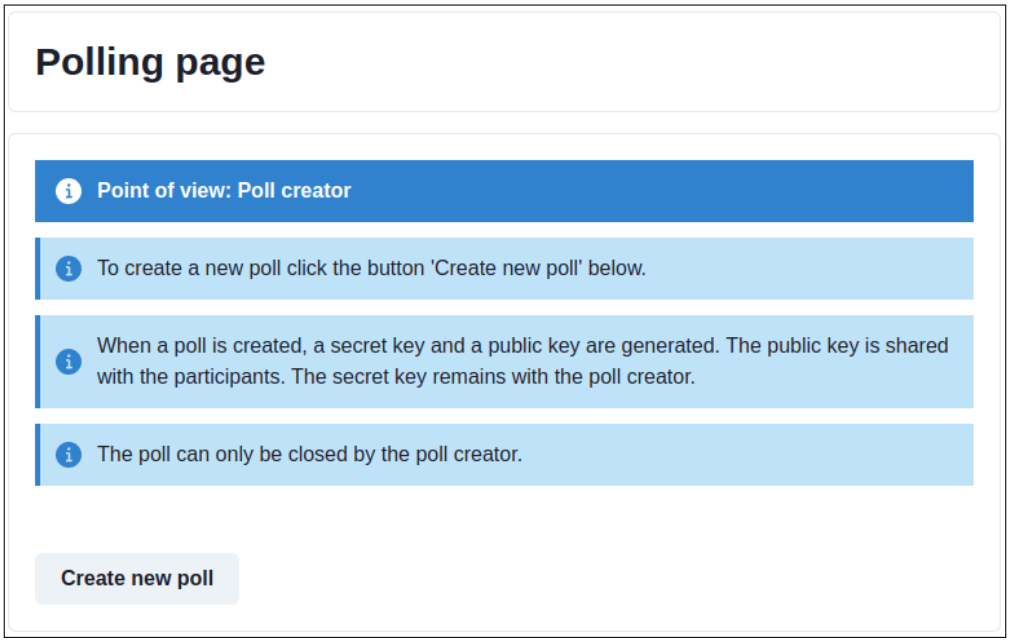

Figure 8: The poll creator's point of view before creating a poll

Creating a new poll After clicking the **Start** button the actual polling page is loaded. At first, it looks like Fig. 8. From now on, information about which role the user is in will be displayed at the top of the page (point of view). The first role is that of the poll creator. The user is asked to create a new poll by clicking the corresponding button.

The first participant After clicking the **Create new poll** button the point of view changes to the first participant of the poll. A table appears in the middle of the page where the participants can make their entries, see Fig. 9 on the following page. Additional information about the poll is given to the user at the top of the page. At the bottom of the page there is information about how many participants can still take part in the poll. This restriction is due to the chosen parameters of the BFV encryption scheme, see Section 3.3 on page 46.

The first attendee can now enter their availability by clicking on the check boxes below the weekdays. An example is shown in Fig. 9 on the following page. Clicking the **Submit** button converts the input into a binary vector and encrypts it as described in Section 3.1 on page 38.

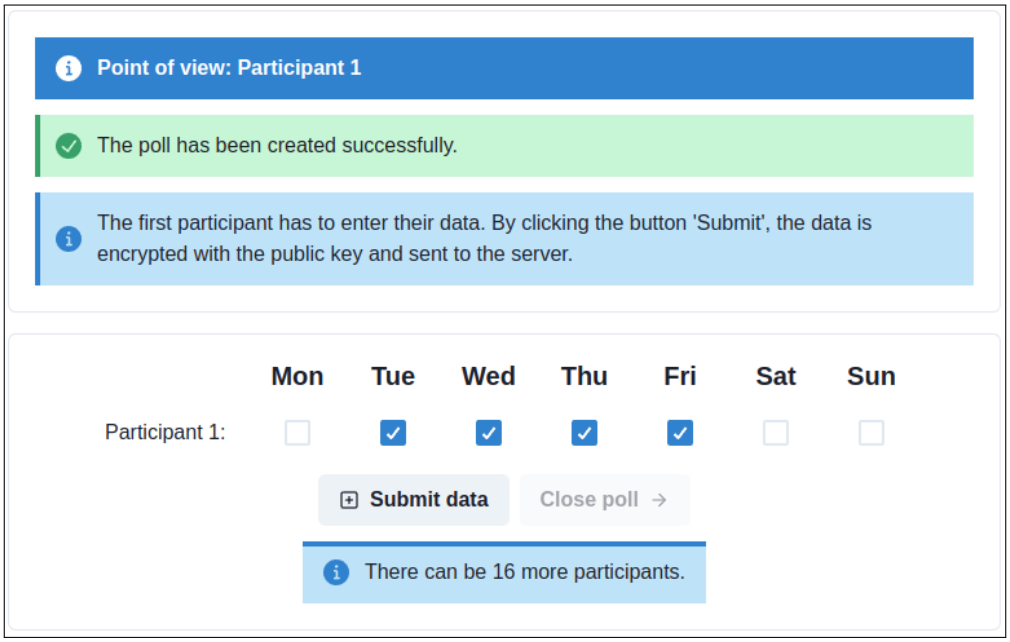

Figure 9: The first participant's point of view

The next participant After an entry has been made and sent, the page updates again. New information about the polling process appears at the top of the page. The user's role changes to the next participant. A new row of check boxes is displayed for this participant. The check boxes from the previous row are replaced with lock icons to symbolize that the answers of the last participants have been encrypted and are therefore no longer visible. This can be seen in Fig. 10 on the following page.

Closing the poll When at least one response to the poll has been received, the **Close poll** button is enabled. Clicking this button closes the poll and triggers the server to calculate the result.

To complete the example, another possible assignment is shown in Fig. 10 on the next page. If this is then submitted and the poll is then closed, the result is calculated and displayed, as can be seen in Fig. 11 on page 53.

The result of the poll is color coded in the last row of the table: A red box indicates that at least one participant does not have time on that day. A green box, on the other hand, indicates that all participants are free on that day. Afterwards, another poll can be made by pressing the **Restart poll** button.

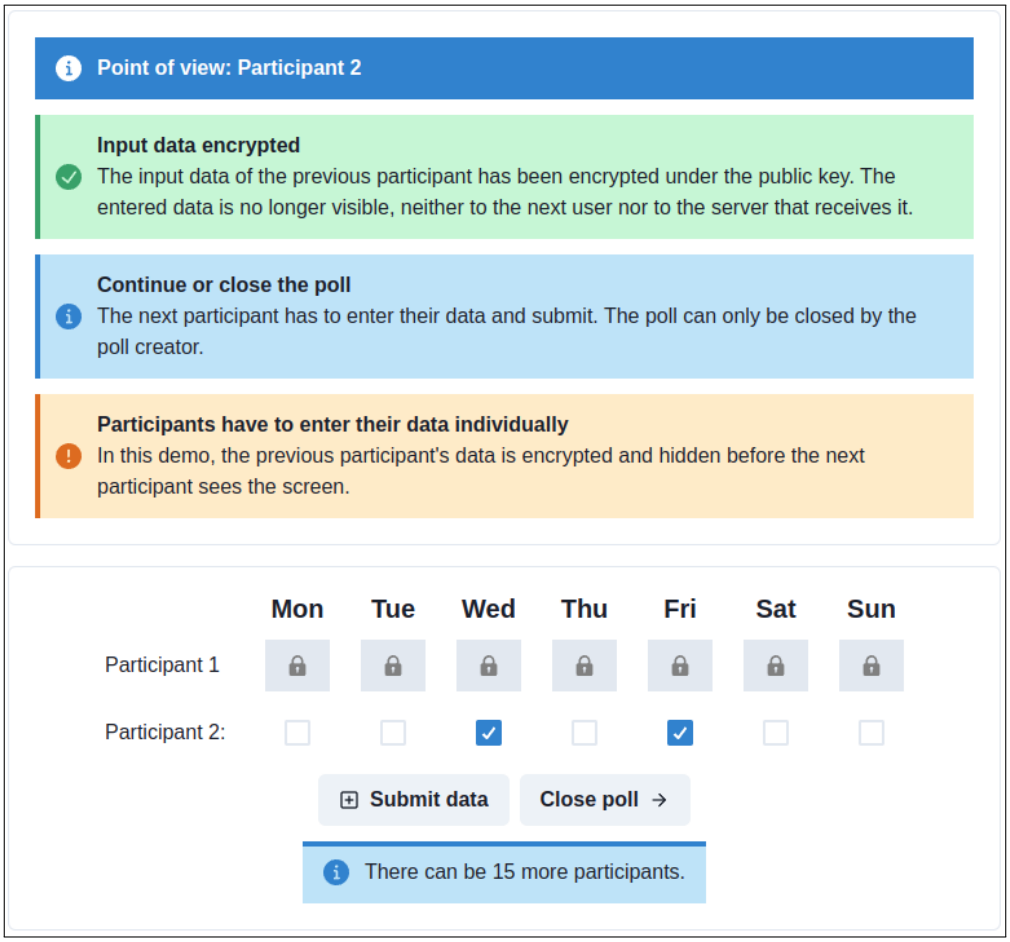

Figure 10: The second participant's point of view

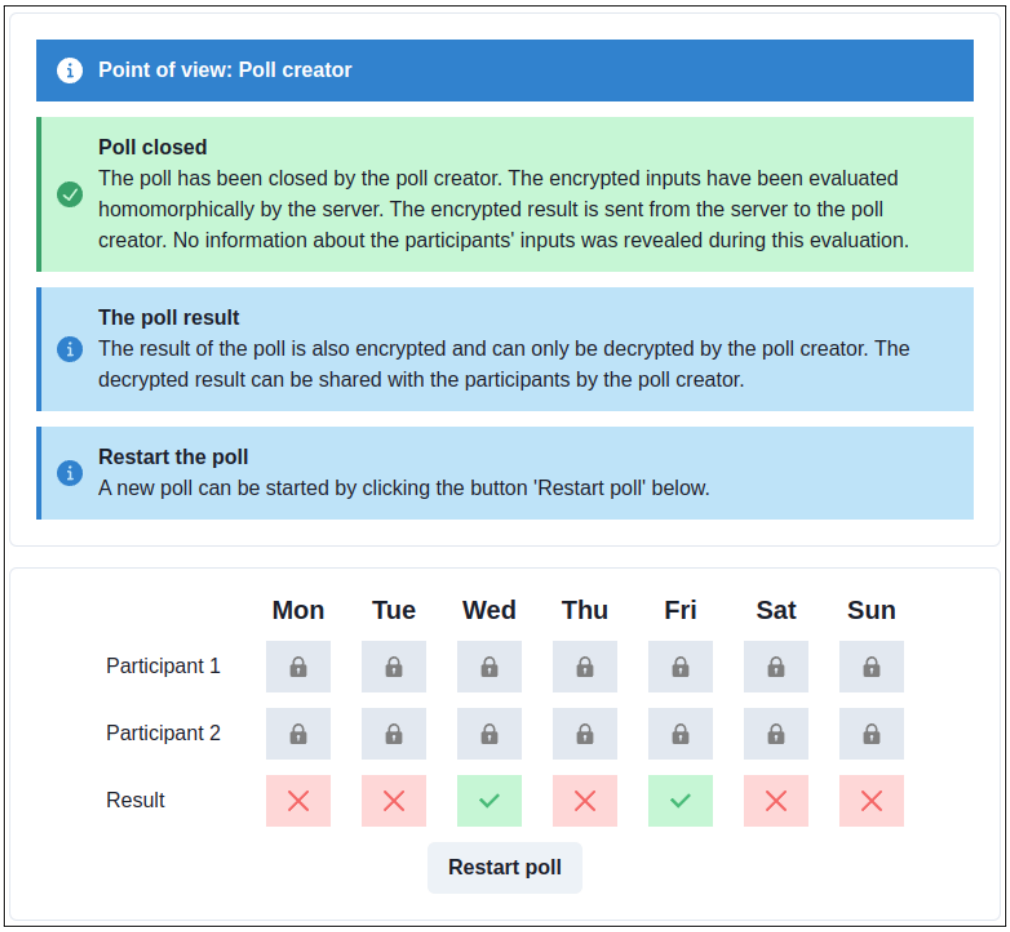

Figure 11: The poll creator's point of view after the poll is closed

3.5 Contribution to CrypTool-Online

The application described in Chapter 3 will be integrated as a plugin into the CrypTool-Online (CTO) project. „CrypTool-Online (CTO for short) offers applications for testing, learning and discovering old and modern cryptography.“ [17] A new version of the website is under development where this application will be made available.[31]

To contribute to the CTO project, a fork of the main repository on GitHub was created so that one can work on their own content in an isolated environment. This way one can either provide entire plugins or help improve existing plugins. In order to merge your changes back into the original repository you can create a pull request. All programming was done in Visual Studio Code.[32] The editor is shown in Fig. 12 on the next page.

Detailed and easy-to-understand documentation summarizes all the steps you need to take to participate. Links to the documentation of the used libraries and frameworks used help to understand and use them very quickly. The most important ones are `Next.js`, `React`, `Webpack` and `Chakra-UI`.

The structure of CTO is based on plugins, which are simply organized in folders. This folder structure is then reflected in the user interface. For example, the application developed in this thesis is accessed via the following folder path: `/ctoapps/he-poll`. In addition, the overview of all plugins on the home page is clearly categorized and provided with short descriptions via a file called `ctoapps.json`.

[31] The current development status can be seen at `https://dev.cryptool.org/`.
Update: In the meantime, this web app is part of the productive CTO: see `https://www.cryptool.org/en/cto/he-poll/`.

[32] `https://code.visualstudio.com`

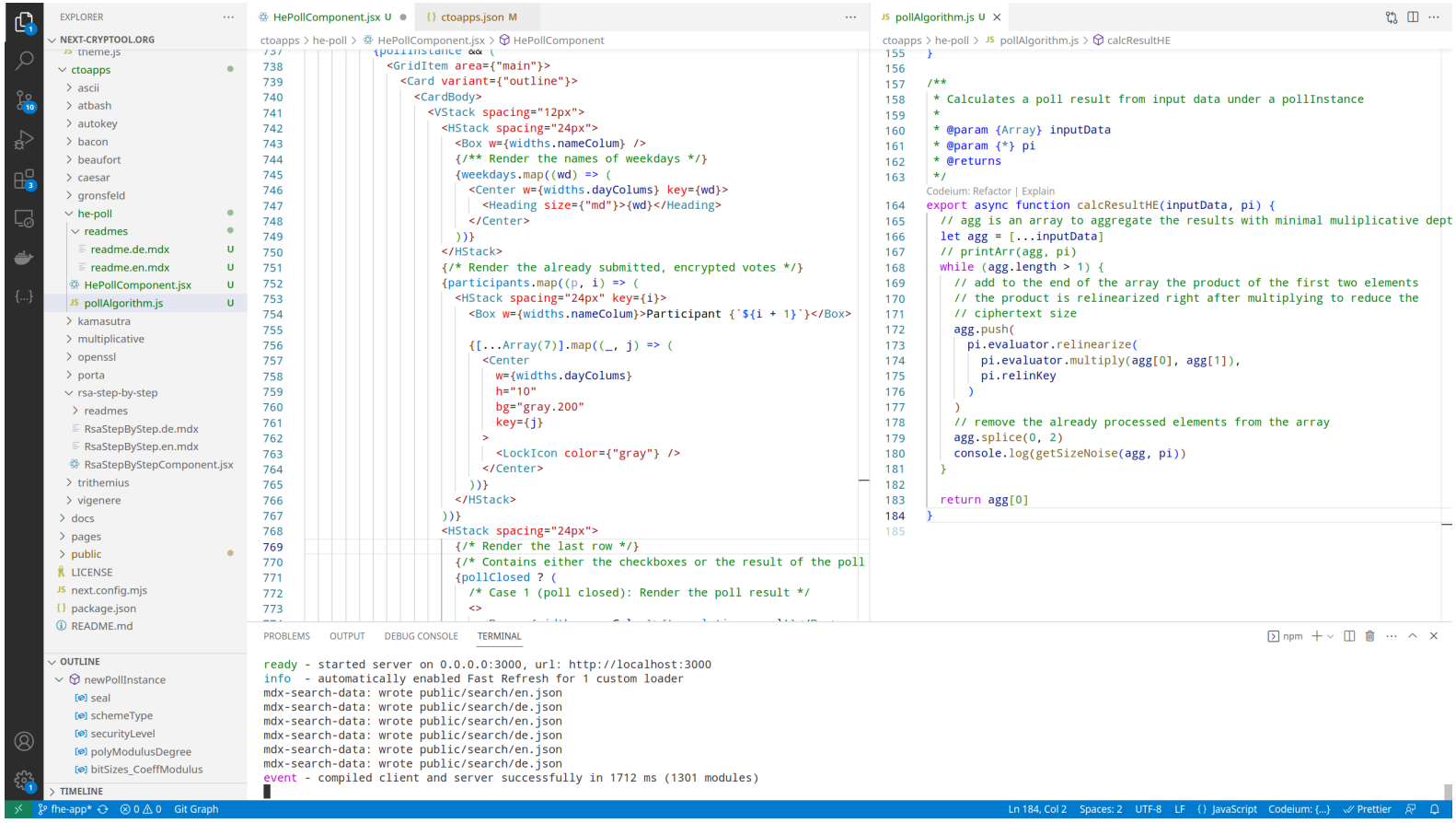

Figure 12: Visual Studio Code

4 Evaluation and conclusion

This chapter summarizes the findings of this thesis. Then, a conclusion is drawn that relates to the title of the paper.

The theory is very complex In summary, while the basic concept of homomorphic encryption (HE) could be made rather easy to understand, fully homomorphic encryption (FHE) cannot be explained in depth in simple words. The main reason for this is the inconsistent notation of parameters in the literature and their extremely complex intertwining in the theory of each scheme (see Section 2.4). The assumptions on which the security of FHE is based on require a broad fundamental knowledge in various branches of mathematics, such as lattices, probability theory but also, of course, cryptography. These are rather high theoretical requirements for a bachelor's thesis in computer science.

Relatively simple implementation Against this background, the implementation of the application for CrypTool-Online (CTO) was easier than expected, although not trivial. The main idea, based on the Lattigo polls demo, was easy to extract. The evaluation algorithm had to be transferred from Go to JS and then implemented with node-seal instead of Lattigo. The very well documented libraries of Lattigo and node-seal made the latter more comfortable. The early problem of having little programming experience in both Go and JS was excellently solved with Codeium's[33] explain feature (see Fig. 14 on the following page).

The architecture of our application can be seen in Fig. 13 on the next page. The implementation of the GUI in JS with Next.js[34], React[35] and Chakra-UI[36] was also supported by very good library documentation of the mentioned software. Surprisingly, using the ChatGPT[37] chat bot to quickly turn concrete ideas into code has proven useful here as well. The used libraries shown in Fig. 13 are SEAL and node-seal. node-seal uses Wasm to access the SEAL library and is imported as a JS library.

[33] Codeium is „a free AI-powered toolkit“, which is also available for Visual Studio Code `https://codeium.com`

[34] `https://nextjs.org`

[35] `https://react.dev`

[36] `https://chakra-ui.com`

[37] `https://chat.openai.com`

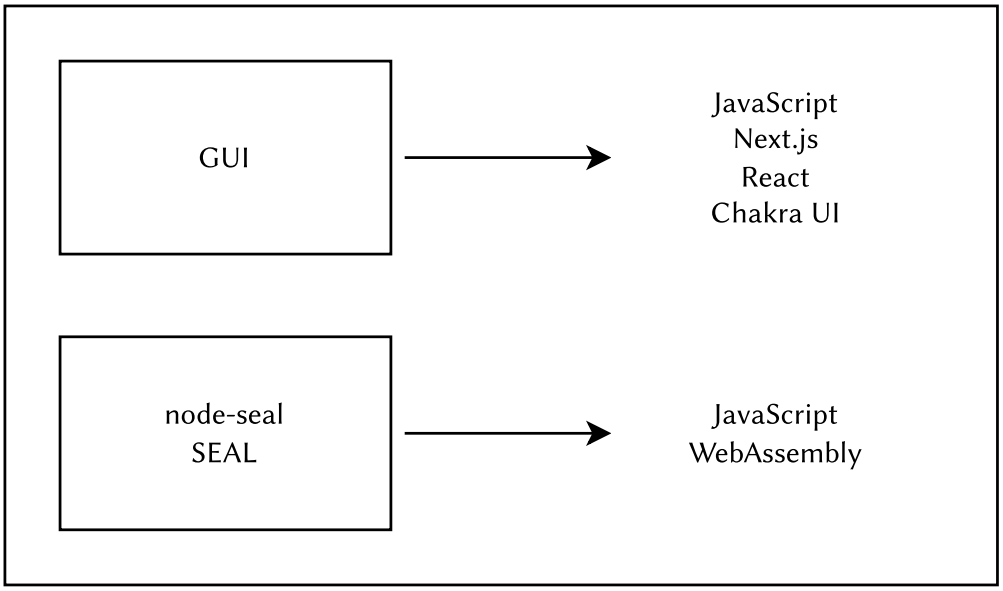

Figure 13: Architecture of our application

Figure 14: Codeium can explain the whole function or just a line of code

Conclusion: The state of HE and FHE The state of HE consists of constant rapid development, both in theory and in practice. This is exemplified by the fact that the latest HE standard [4] has not even included the CKKS scheme [14] (see Section 2.3) nor the RNS optimization [8] (see Section 2.6), although both are already available in most libraries, e.g., SEAL [62] and Lattigo. [39] One can also see it in the fact that since 2018 there is more than one publication per week that has FHE in its title.[38] See also Fig. 1.

Nevertheless, FHE is still far from being practically relevant. First of all, this is due to the extensive theoretical knowledge, mentioned in Section 2.4, that a programmer must have in order to program a meaningful application. In addition, current methods are still very slow and require a lot of memory and processing power. [52, pp. 58-60]

In practice this problem is addressed by also using **somewhat** homomorphic encryption (SHE) schemes instead of FHE schemes, see Section 2.4.1.

[38] https://dblp.org/search/publ?q=fully+homomorphic+encryption

Bibliography

[1] *About Emscripten*. May 25, 2023. URL: https://emscripten.org/docs/introducing_emscripten/about_emscripten.html (cit. on p. 45).

[2] Shweta Agrawal, Shafi Goldwasser, and Saleet Mossel. „Deniable Fully Homomorphic Encryption from Learning with Errors“. In: *Advances in Cryptology – CRYPTO 2021*. Ed. by Tal Malkin and Chris Peikert. Cham: Springer International Publishing, 2021, pp. 641–670. ISBN: 978-3-030-84245-1 (cit. on p. 6).

[3] Adi Akavia, Craig Gentry, Shai Halevi, and Margarita Vald. *Achievable CCA2 Relaxation for Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2022/282. 2022. URL: https://eprint.iacr.org/2022/282 (cit. on p. 6).

[4] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. *Homomorphic Encryption Security Standard*. Tech. rep. Toronto, Canada: HomomorphicEncryption.org, Nov. 2018. URL: http://homomorphicencryption.org/wp-content/uploads/2018/11/HomomorphicEncryptionStandardv1.1.pdf (cit. on pp. 8, 29, 33, 34, 37, 58).

[5] Nick Angelou. Apr. 21, 2023. URL: https://github.com/s0l0ist/node-seal (cit. on pp. 26, 45).

[6] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. „Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE“. In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 483–501. ISBN: 978-3-642-29011-4 (cit. on p. 36).

[7] Thomas Attema, Pedro Capitão, and Lisa Kohl. „On Homomorphic Secret Sharing from Polynomial-Modulus LWE“. In: *Public-Key Cryptography – PKC 2023*. Ed. by Alexandra Boldyreva and Vladimir Kolesnikov. Cham: Springer Nature Switzerland, 2023, pp. 3–32. ISBN: 978-3-031-31371-4 (cit. on p. 6).

[8] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. „A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes“. In: *Selected Areas in Cryptography – SAC 2016*. Ed. by Roberto Avanzi and Howard Heys. Cham: Springer International Publishing, 2017, pp. 423–442 (cit. on pp. 6, 15, 28, 37, 58).

[9] Josh Benaloh. „Dense Probabilistic Encryption“. In: *Proceedings of the workshop on selected areas of cryptography*. 1994, pp. 120–128. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/1999/02/dpe.pdf (cit. on p. 14).

[10] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. *FINAL: Faster FHE instantiated with NTRU and LWE*. Cryptology ePrint Archive, Paper 2022/074. 2022. URL: https://eprint.iacr.org/2022/074 (cit. on p. 6).

[11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. *Fully Homomorphic Encryption without Bootstrapping*. Cryptology ePrint Archive, Paper 2011/277. 2011. URL: https://eprint.iacr.org/2011/277 (cit. on pp. 6, 18, 19).

[12] Johannes Buchmann. *Einführung in die Kryptographie*. 6th ed. Springer, 2016. URL: https://doi.org/10.1007/978-3-642-39775-2 (cit. on pp. 9, 25).

[13] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. „Efficient Multi-Key Homomorphic Encryption with Packed Ciphertexts with Application to Oblivious Neural Network Inference“. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 395–412. ISBN: 9781450367479. URL: https://doi.org/10.1145/3319535.3363207 (cit. on p. 36).

[14] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. „Homomorphic Encryption for Arithmetic of Approximate Numbers“. In: *International conference on the theory and application of cryptology and information security*. Springer. 2017, pp. 409–437 (cit. on pp. 6, 19, 58).

[15] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. „Fast Fully Homomorphic Encryption Over the Torus“. In: *Journal of Cryptology* 33.1 (2020), pp. 34–91 (cit. on pp. 6, 19, 20).

[16] Orel Cosseron, Clément Hoffmann, Pierrick Méaux, and François-Xavier Standaert. *Towards Globally Optimized Hybrid Homomorphic Encryption – Featuring the Elisabeth Stream Cipher*. Cryptology ePrint Archive, Paper 2022/180. 2022. URL: https://eprint.iacr.org/2022/180 (cit. on p. 6).

[17] *CrypTool-Online*. May 26, 2023. URL: https://www.cryptool.org/en/cto/ (cit. on p. 54).

[18] *DeepL Translator*. URL: https://www.deepl.com/translator (cit. on p. 7).

[19] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. „Fully Homomorphic Encryption over the Integers“. In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2010, pp. 24–43 (cit. on pp. 6, 8, 18, 19).

[20] Léo Ducas and Daniele Micciancio. „FHEW: Bootstrapping Homomorphic Encryption in less than a second“. In: *Advances in Cryptology–EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I 34*. Springer. 2015, pp. 617–640 (cit. on pp. 6, 19, 20).

[21] T. Elgamal. „A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms“. In: *IEEE Transactions on Information Theory* 31.4 (1985), pp. 469–472. DOI: 10.1109/TIT.1985.1057074 (cit. on p. 14).

[22] Bernhard Esslinger, ed. *Das CrypTool-Buch: Kryptographie lernen und anwenden mit CrypTool und SageMath*. 12th ed. CrypTool-Projekt, 2018 (cit. on pp. 8, 12).

[23] Junfeng Fan and Frederik Vercauteren. „Somewhat Practical Fully Homomorphic Encryption“. In: *Cryptology ePrint Archive* (2012) (cit. on pp. 6, 15, 16, 19, 25, 26, 28, 36).

[24] David Froelicher, Juan Ramón Troncoso-Pastoriza, Apostolos Pyrgelis, Sinem Sav, Joao Sa Sousa, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. „Scalable Privacy-Preserving Distributed Learning“. In: *Proc. Priv. Enhancing Technol.* 2021.2 (2021), pp. 323–347. URL: https://doi.org/10.2478/popets-2021-0030 (cit. on pp. 7, 21).

[25] Robin Geelen, Ilia Iliashenko, Jiayi Kang, and Frederik Vercauteren. *On Polynomial Functions Modulo p^e and Faster Bootstrapping for Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2022/1364. 2022. URL: https://eprint.iacr.org/2022/1364 (cit. on p. 6).

[26] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. Stanford University, 2009 (cit. on pp. 6, 8, 14–16, 18, 19, 37).

[27] Craig Gentry. „Computing Arbitrary Functions of Encrypted Data“. In: *Commun. ACM* 53.3 (Mar. 2010), pp. 97–105. ISSN: 0001-0782. URL: https://doi.org/10.1145/1666420.1666444 (cit. on pp. 8, 17).

[28] Craig Gentry. „Fully Homomorphic Encryption Using Ideal Lattices“. In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 2009, pp. 169–178 (cit. on p. 6).

[29] Craig Gentry and Shai Halevi. „Implementing Gentry's Fully-Homomorphic Encryption Scheme“. In: *Advances in Cryptology–EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings 30*. Springer. 2011, pp. 129–148 (cit. on pp. 6, 19).

[30] Craig Gentry, Shai Halevi, and Nigel P. Smart. „Homomorphic Evaluation of the AES Circuit“. In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 850–867. ISBN: 978-3-642-32009-5 (cit. on p. 16).

[31] Craig Gentry, Amit Sahai, and Brent Waters. „Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based“. In: *Cryptology ePrint Archive* (2013). URL: https://eprint.iacr.org/2013/340 (cit. on pp. 6, 19, 20).

[32] Shafi Goldwasser and Silvio Micali. „Probabilistic Encryption & How to Play Mental Poker Keeping Secret All Partial Information“. In: STOC '82. San Francisco, California, USA: Association for Computing Machinery, 1982, pp. 365–377. URL: https://doi.org/10.1145/800070.802212 (cit. on p. 14).

[33] Jincheol Ha, Seongkwang Kim, Byeonghak Lee, Jooyoung Lee, and Mincheol Son. *Rubato: Noisy Ciphers for Approximate Homomorphic Encryption (Full Version)*. Cryptology ePrint Archive, Paper 2022/537. 2022. URL: https://eprint.iacr.org/2022/537 (cit. on p. 6).

[34] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. „Bringing the Web up to Speed with WebAssembly“. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 185–200. ISBN: 9781450349888. URL: https://doi.org/10.1145/3062341.3062363 (cit. on p. 45).

[35] *Homomorphic Encryption Sandbox*. Feb. 5, 2024. URL: https://s0l0ist.github.io/seal-sandbox/ (cit. on p. 26).

[36] Marc Joye. „SoK: Fully Homomorphic Encryption over the [Discretized] Torus“. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.4 (Aug. 2022), pp. 661–692. URL: https://tches.iacr.org/index.php/TCHES/article/view/9836 (cit. on p. 6).

[37] Markulf Kohlweiss, Anna Lysyanskaya, and An Nguyen. *Privacy-Preserving Blueprints*. Cryptology ePrint Archive, Paper 2022/1536. 2022. URL: https://eprint.iacr.org/2022/1536 (cit. on p. 6).

[38] Lausanne Laboratory for Data Security. May 3, 2023. URL: https://github.com/ldsec/lattigo-polls-demo (cit. on pp. 38, 45, 48).

[39] *Lattigo v4*. EPFL-LDS, Tune Insight SA. Aug. 2022. URL: https://github.com/tuneinsight/lattigo (cit. on pp. 45–47, 58).

[40] Kang Hoon Lee and Ji Won Yoon. „Discretization Error Reduction for High Precision Torus Fully Homomorphic Encryption“. In: *Public-Key Cryptography – PKC 2023*. Ed. by Alexandra Boldyreva and Vladimir Kolesnikov. Cham: Springer Nature Switzerland, 2023, pp. 33–62. ISBN: 978-3-031-31371-4 (cit. on p. 6).

[41] Yongwoo Lee, Joon-Woo Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and HyungChul Kang. „High-Precision Bootstrapping for Approximate Homomorphic Encryption by Error Variance Minimization“. In: *Advances in Cryptology – EUROCRYPT 2022*. Ed. by Orr Dunkelman and Stefan Dziembowski. Cham: Springer International Publishing, 2022, pp. 551–580. ISBN: 978-3-031-06944-4 (cit. on p. 6).

[42] Yongwoo Lee, Daniele Micciancio, Andrey Kim, Rakyong Choi, Maxim Deryabin, Jieun Eom, and Donghoon Yoo. *Efficient FHEW Bootstrapping with Small Evaluation Keys, and Applications to Threshold Homomorphic Encryption*. Cryptology ePrint Archive, Paper 2022/198. 2022. URL: https://eprint.iacr.org/2022/198 (cit. on p. 6).

[43] Baiyu Li, Daniele Micciancio, Mark Schultz, and Jessica Sorrell. *Securing Approximate Homomorphic Encryption Using Differential Privacy*. Cryptology ePrint Archive, Paper 2022/816. 2022. URL: https://eprint.iacr.org/2022/816 (cit. on p. 6).

[44] *libshe*. June 3, 2023. URL: https://github.com/bogdan-kulynych/libshe (cit. on p. 8).

[45] Feng-Hao Liu and Han Wang. „Batch Bootstrapping I:“ in: *Advances in Cryptology – EUROCRYPT 2023*. Ed. by Carmit Hazay and Martijn Stam. Cham: Springer Nature Switzerland, 2023, pp. 321–352. ISBN: 978-3-031-30620-4 (cit. on p. 6).

[46] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. *Large-Precision Homomorphic Sign Evaluation using FHEW/TFHE Bootstrapping*. Cryptology ePrint Archive, Paper 2021/1337. 2021. URL: https://eprint.iacr.org/2021/1337 (cit. on p. 6).

[47] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. „On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption“. In: *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*. STOC ’12. New York, New York, USA: Association for Computing Machinery, 2012, pp. 1219–1234. ISBN: 9781450312455. URL: https://doi.org/10.1145/2213977.2214086 (cit. on p. 36).

[48] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. „On Ideal Lattices and Learning with Errors over Rings“. In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23. ISBN: 978-3-642-13190-5 (cit. on p. 24).

[49] Microsoft. *1_bfv_basics.cpp*. Microsoft Research, Redmond, WA. May 22, 2023. URL: https://github.com/microsoft/SEAL/blob/main/native/examples/1_bfv_basics.cpp (cit. on pp. 23, 24, 35).

[50] Christian Mouchet, Juan Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. *Multiparty Homomorphic Encryption from Ring-Learning-With-Errors*. Cryptology ePrint Archive, Paper 2020/304. 2020. URL: https://eprint.iacr.org/2020/304 (cit. on p. 36).

[51] Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. „Lattigo: a Multiparty Homomorphic Encryption Library in Go“. In: *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*. CONF. 2020, pp. 64–70 (cit. on pp. 6, 7, 21, 36, 38, 40).

[52] Jörn Müller-Quade, Thomas Agrikola, Laurin Benz, and Anne Müller. „Encrypted Computing Compass“. In: (Nov. 2022). URL: https://www.cyberagentur.de/encrypted-computing-compass/ (cit. on pp. 7, 12, 13, 18–20, 37, 40, 58).

[53] Karthik Nandakumar, Nalini Ratha, Sharath Pankanti, and Shai Halevi. „Towards Deep Neural Network Training on Encrypted Data“. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2019, pp. 40–48. DOI: 10.1109/CVPRW.2019.00011 (cit. on p. 7).

[54] Pascal Paillier. „Public-Key Cryptosystems Based on Composite Degree Residuosity Classes“. In: *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*. EUROCRYPT'99. Prague, Czech Republic: Springer-Verlag, 1999, pp. 223–238. ISBN: 3540658890 (cit. on pp. 12, 14).

[55] Rachel Player. „Parameter selection in lattice-based cryptography“. PhD thesis. Royal Holloway, University of London, Egham, UK, 2018. URL: https://pure.royalholloway.ac.uk/ws/files/29983580/2018playerrphd.pdf (cit. on p. 24).

[56] Oded Regev. „On Lattices, Learning with Errors, Random Linear Codes, and Cryptography“. In: *J. ACM* 56.6 (Sept. 2009). URL: https://doi.org/10.1145/1568318.1568324 (cit. on p. 14).

[57] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. „On Data Banks and Privacy Homomorphisms“. In: *Foundations of secure computation* 4.11 (1978), pp. 169–180 (cit. on p. 14).

[58] Ronald L Rivest, Adi Shamir, and Leonard Adleman. „A Method for Obtaining Digital Signatures and Public-Key Cryptosystems“. In: *Communications of the ACM* 21.2 (1978), pp. 120–126 (cit. on pp. 8, 14).

[59] Lawrence Roy and Jaspal Singh. „Large Message Homomorphic Secret Sharing from DCR and Applications“. In: *Advances in Cryptology – CRYPTO 2021*. Ed. by Tal Malkin and Chris Peikert. Cham: Springer International Publishing, 2021, pp. 687–717. ISBN: 978-3-030-84252-9 (cit. on p. 6).

[60] Boaz Sapir. *Data Science without Seeing Data: How to Set Microsoft Open Source SEAL Parameters*. May 19, 2023. URL: https://medium.com/intuit-engineering/data-science-without-seeing-data-how-to-set-microsoft-open-source-seal-parameters-72929b184058 (cit. on p. 27).

[61] Sinem Sav, Apostolos Pyrgelis, Juan Ramón Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux. „POSEIDON: Privacy-Preserving Federated Neural Network Learning“. In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. URL: https://www.ndss-symposium.org/ndss-paper/poseidon-privacy-preserving-federated-neural-network-learning/ (cit. on pp. 7, 21).

[62] *Microsoft SEAL (release 4.1)*. Microsoft Research, Redmond, WA. Jan. 2023. URL: https://github.com/Microsoft/SEAL (cit. on p. 58).

[63] Douglas R. Stinson. *Cryptography – Theory and Practice*. 3rd ed. Chapman & Hall/CRC, 2006 (cit. on p. 8).

[64] Cormen Thomas H, E Charles, Rivest Ronald L, and Stein Clifford. *Introduction To Algorithms Third Edition*. Mit Press, 2009 (cit. on p. 19).

[65] Vinod Vaikuntanathan. *Homomorphic Encryption References*. May 17, 2023. URL: https://people.csail.mit.edu/vinodv/FHE/FHE-refs.html (cit. on pp. 18, 19).

List of Abbreviations

API	application programming interface
AES	Advanced Encryption Standard
AGCD	approximate gcd
BDDP	bounded distance decoding problem
BFV	Brakerski/Fan-Vercauteren
BGV	Brakerski-Gentry-Vaikuntanathan
CKKS	Cheon-Kim-Kim-Song
CRT	chinese remainder theorem
CTO	CrypTool-Online
CVP	closest vector problem
DGHV	Dijk-Gentry-Halevi-Vaikuntanathan
FHE	fully homomorphic encryption
GSW	Gentry-Sahai-Waters
GUI	graphical user interface
HE	homomorphic encryption
HTTPS	Hypertext Transfer Protocol Secure
JS	JavaScript
LWE	learning with errors
MHE	multi-party homomorphic encryption
MPC	multi-party communication
NIST	National Institute of Standards and Technology
PHE	partially homomorphic encryption
PKE	public-key encryption
RLWE	ring learning with errors
RNS	residue number system
RSA	Rivest-Shamir-Adleman
SEAL	Simple Encrypted Arithmetic Library
SHE	somewhat homomorphic encryption
SIMD	single instruction, multiple data
SSSP	sparse subset sum problem
TFHE	Fully Homomorphic Encryption over the Torus
Wasm	WebAssembly
XOR	exclusive or

List of Tables

List of Figures

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, insbesondere keine anderen als die angegebenen Informationen aus dem Internet. Diejenigen Paragraphen der für mich geltenden Prüfungsordnungen, die etwaige Betrugsversuche betreffen, habe ich zur Kenntnis genommen.

Der Speicherung meiner Bachelorarbeit zum Zweck der Plagiatsprüfung stimme ich zu. Ich versichere, dass die elektronische Version mit der gedruckten Version inhaltlich übereinstimmt.

_____ _____

(Datum) (Unterschrift)

Content of the CD

- Bachelor's thesis as PDF
- Source code of the application