

# Bootstrapping fully homomorphic encryption over the integers in less than one second

Hilder Vitor Lima Pereira

University of Luxembourg  
April 2020

**Abstract.** One can bootstrap (R)LWE-based fully homomorphic encryption (FHE) schemes in less than one second, but bootstrapping AGCD-based FHE schemes, also known as FHE over the integers, is still very slow. In this work we propose fast bootstrapping methods for FHE over the integers, closing thus this gap between these two types of schemes. We use a variant of the AGCD problem to construct a new GSW-like scheme that can natively encrypt polynomials, then, we show how the single-gate bootstrapping method proposed by Ducas and Micciancio (EUROCRYPT 2015) can be adapted to FHE over the integers using our scheme, and we implement a bootstrapping that runs in less than one second in a common personal computer. We also perform multi-value bootstrapping on non-binary message spaces achieving running times close to one second with modest memory requirements.

**Keywords:** Fully Homomorphic Encryption, AGCD, Bootstrapping.

## 1 Introduction

The two main families of fully homomorphic encryption (FHE) schemes are the ones based on lattices, mainly on the Ring Learning with Errors (RLWE) problem, and the schemes over the integers, based on the Approximate Greatest Common Divisor (AGCD) problem. Immediately after the first FHE scheme was proposed by Gentry [Gen09], a scheme over the integers was put forth as a simpler alternative [DGHV10]. Thereafter, several techniques were proposed to improve the efficiency of FHE, and one always found ways to apply those techniques to both families of homomorphic schemes. For example, a method to reduce the noise by scaling a ciphertext and switching the modulus of the ciphertext space, known as modulus switching, was proposed in [BV11] and was soon adapted for schemes over the integers [CNT12]. A technique known as batching, which consists in encrypting several messages into a single ciphertext so that each homomorphic operation acts in parallel on all the encrypted messages, has also been applied to RLWE schemes [BGV12,GHS12] and to schemes over the integers [CCK<sup>+</sup>13]. Finally, in 2013, Gentry, Sahai, and Waters introduced a FHE scheme that uses a decomposition technique to turn the noise growth of homomorphic products roughly additive [GSW13], i.e., the homomorphic product of

two ciphertexts  $c$  and  $c'$  yields a ciphertext  $c_{mult}$  whose noise is approximately the noise of  $c$  plus the noise of  $c'$ . Even this technique was adapted to the schemes over the integers [BBL17].

However, new fast bootstrapping techniques, one of the last great achievements of FHE, has only been available for (R)LWE schemes: In [ASP14], it was proposed to bootstrap a base scheme whose ciphertext space is  $\mathbb{Z}_q$  by using a GSW-like scheme whose plaintext space contains  $\mathbb{Z}_q$ . Because of the slow noise growth of GSW-like schemes, the final noise accumulated in the refreshed ciphertext is only polynomial in the security parameter  $\lambda$ , therefore, it is not necessary to set large parameters for the base scheme as it was done in previous bootstrapping methods, where the parameters have to allow a scheme to evaluate its own decryption function. Then, in [DM15], the authors found an efficient way to represent  $\mathbb{Z}_q$ , removed the expensive final step of the method proposed in [ASP14], and implemented a bootstrapping that runs in less than one second in a common laptop using a GSW-like scheme based on the RLWE. The running times of [DM15] were further improved in [CGGI16] and a base scheme based on LWE was bootstrapped in less than 0.1 second also using a RLWE-based GSW-like scheme. Nevertheless, none of those techniques has been adapted to FHE over the integers.

The main difficulties one has to deal with when trying to create similar bootstrapping methods for FHE over the integers are:

1. One needs an efficient GSW-like scheme based on the AGCD problem. For instance, the GSW-like scheme proposed in [BBL17] is far from practical. The scheme of [Per20] has better running times and, at first glance, seems to be a good choice, however, the size of the bootstrapping keys that it produces is huge.
2. The modulus  $p$  is secret: the decryption function of (R)LWE-based schemes is defined modulo a *public* integer  $q$ , thus, all the homomorphic operations performed during the bootstrapping can safely disclose  $q$ , but for AGCD-based schemes, we have an integer  $p$  which is at the same time the modulus and the secret key, hence, the bootstrapping must hide  $p$ .
3. The modulus  $p$  is exponentially large in  $\lambda$ : in (R)LWE-based schemes, the modulus  $q$  is just polynomially large in the security parameter, while in FHE over the integers we have  $p \in \Omega(2^\lambda)$ , thus, the techniques to perform fast bootstrapping require that the message space of the GSW-like scheme contain a set that is exponentially large, i.e., the set  $\mathbb{Z}_p$ .

Thus, in this work, we aim to close the gap between (R)LWE- and AGCD-based schemes by proposing fast bootstrapping methods for FHE over the integers. Namely, we introduce a new hardness problem that cannot be easier than the AGCD problem, then we use it to construct an efficient GSW-like scheme that works homomorphically on polynomial rings of the form  $\mathbb{Z}_t[x]/\langle f \rangle$ . There-with we first show how to perform gate bootstrapping, as in [DM15,CGGI16], and then a more general multi-value bootstrapping, as in [CIM19]. We implemented a proof-of-concept in C++ and refreshed ciphertexts of FHE schemes over the integers in less than one second.

## 1.1 Overview of our techniques and results

*New underlying problem and GSW-like scheme:* Our first contribution is to use the AGCD problem to construct a GSW-like homomorphic encryption scheme that operates efficiently and natively on polynomial rings. We remark that given  $N$  AGCD instances  $c_i := pq_i + r_i$ , one can represent them as a polynomial  $c(x) := \sum_{i=0}^{N-1} c_i x^i$ , which can then be written as  $c(x) = pq(x) + r(x)$ . Thus, if we extend the AGCD problem to sample polynomials  $q(x)$  and  $r(x)$  and return  $pq(x) + r(x)$ , we obtain an equivalent problem. But now, by fixing a polynomial ring  $R$ , for example,  $R = \mathbb{Z}[x]/\langle x^N + 1 \rangle$ , and a secret polynomial  $k(x) \in R$ , we can obtain randomized instances of the form  $(pq(x) + r(x))k(x)$ . Because we are randomizing a problem that is equivalent to the AGCD, we obtain a problem that cannot be easier than the AGCD problem. We call it Randomized (Polynomial) AGCD (RAGCD) problem. Moreover, as it was noticed in [CP19], solving randomized versions of the AGCD problem seems to be harder than solving the original AGCD problem, therefore, we can select smaller parameters. In particular, each AGCD sample is a  $\gamma$ -bit integer, but in our case each coefficient of the polynomials will be an integer with bit length around  $\gamma/N$ , where  $N$  is the degree of  $k(x)$ . Hence, we can use the RAGCD problem to encrypt a degree- $N$  polynomial  $m$  into a degree- $N$  polynomial  $c$  whose total bit-length is then  $N \cdot \gamma/N = \gamma$ , while using the AGCD problem would require one  $\gamma$ -bit ciphertext for each coefficient, resulting in a total of  $N\gamma$  bits.

Thus, using the RAGCD problem, we propose a GSW-like scheme that can encrypt a polynomial  $m \in R$  in two formats:

- Scalar format:  $(pq + r + \alpha m) \cdot k \in R$ .
- Vector format:  $(p\mathbf{q} + \mathbf{r}) \cdot k + \mathbf{g}m \in R^\ell$ , where  $\mathbf{g} = (b^0, \dots, b^{\ell-1})$  for some  $b \in \mathbb{Z}$ .

Therewith we can define an efficient mixed homomorphic multiplication, from  $R \times R^\ell$  to  $R$ , that is somehow equivalent to the external product used in [CGGI16]. We notice that the main source of efficiency of the bootstrapping method proposed in [CGGI16] is the use of this external product, hence, we have then the first piece of a fast bootstrapping for FHE over the integers.

*Fast bootstrapping for FHE over the integers:* Firstly, notice that simply trying to implement the bootstrapping procedures of [DM15] or [CGGI16] with our scheme would not work, since it would require us to use  $N > p \in \Omega(2^\lambda)$ , which is not efficient, and it would leak  $p$ . Therefore, to solve these issues related to the size and the privacy of the modulus used in the decryption of AGCD-based schemes, we propose to perform a “hidden approximate modulus switching”. Namely, given a ciphertext  $c = pq + r + mp/t$  to be bootstrapped, multiplying it by  $N/p$  would switch the modulus, resulting in  $c' = Nq + r' + mN/t$ , for a potentially small  $N$  that could be managed by our scheme. Of course, we cannot do it before refreshing because having access to  $N/p$  would leak  $p$ . Even if we could perform the modulus switching in a secure way, without revealing  $p$ , as in [CNT12], the resulting ciphertext  $c'$  would leak the message  $m$ , because  $N$  is known. Thus, we propose that the product  $c \cdot N/p$  be performed as part of the

refreshing procedure, so that the secret key  $p$  is encrypted in the bootstrapping keys and the resulting ciphertext  $c'$  is only produced in an encrypted form.

Essentially, since  $y := x^2$  has order  $N$  in  $R := \mathbb{Z}[x]/\langle x^N + 1 \rangle$ , we can use our GSW-like scheme, which we name GAHE, to work homomorphically over  $\mathbb{Z}_N$ . Thus, we would like to encrypt messages  $y^{2^i N/p}$  for  $0 \leq i < \gamma$ , and then, to refresh a  $\gamma$ -bit ciphertext  $c = pq + r + mp/t$  of the base scheme, we would decompose  $c$  obtaining  $(c_0, \dots, c_{\gamma-1})$  and use the homomorphic mixed product to compute a GAHE ciphertext  $\tilde{c}$  encrypting  $\prod_{i=0}^{\gamma-1} y^{c_i 2^i N/p \bmod N} = y^{r' + mN/t}$ . After this, we could apply techniques similar to those of [DM15] to transform  $\tilde{c}$  into a base scheme ciphertext encrypting  $m$ . The problem now is that  $2^i N/p$  is not integer. Hence, we encrypt  $y^{\lfloor 2^i N/p \rfloor}$  instead. By noticing that  $\lfloor 2^i N/p \rfloor = 2^i N/p + \epsilon_i$  for some  $\epsilon_i \in [-1/2, 1/2]$ , we see that computing the same sequence of homomorphic products yields a GAHE encryption of  $y^{r' + mN/t + \epsilon}$  for some term  $\epsilon$  that is not too big. Finally, we propose a functional key-switching to transform this GAHE ciphertext into a base scheme (AGCD-based) encryption of  $m$ . By choosing the parameters carefully, the noise term of the final ciphertext is smaller than the initial noise.

*More general multi-value bootstrapping:* Finally, we adapt our bootstrapping procedure to a more general scenario in which a ciphertext of the base scheme has the form  $(pq + r + \alpha m)k \in R$ , that is, it is also a polynomial, and the message  $m$  belongs to  $\mathbb{Z}_t$  instead of  $\{0, 1\}$ . We do so by using our functional key-switching to securely transform polynomial ciphertexts encrypted under the RAGCD problem into integer ciphertexts based on the AGCD problem. From this point on, we can basically apply our original bootstrapping, but taking into account that the message space is now  $\mathbb{Z}_t$ .

*Implementation and practical results:* We implemented our bootstrapping procedures in C++ and executed experiments similar to [DM15], [CGGI16], and [CIM19]. Although our implementation is not optimized, we obtained running times and memory consumption similar to [DM15], i.e., we could bootstrap the base scheme in less than one second. For the best of our knowledge, all the previous bootstrapping methods for FHE over the integers took several seconds (or even minutes). As for the multi-value bootstrapping, we could execute it in about the same time of [CIM19] and using approximately half of the memory. All the details are shown in Section 7.

## 2 Theoretical background and related works

### 2.1 Notation and basic facts

We use  $R$  to denote the cyclotomic ring  $\mathbb{Z}[x]/\langle x^N + 1 \rangle$ , where  $N$  is a power of two. When we refer to an element  $f$  of  $R$ , we always mean the unique representative of degree smaller than  $N$ , thus, writing  $f = \sum_{i=0}^{N-1} f_i x^i$  is unambiguous and we can define the coefficient vector of  $f$  as  $\phi(f) := (f_0, \dots, f_{N-1})$ . The anti-circulant matrix of  $f$  is the matrix  $\Phi(f) \in \mathbb{Z}^{N \times N}$  such that the  $i$ -th row is equal

to  $\phi(x^{i-1}f)$  for  $1 \leq i \leq N$ . It is worth noticing that for  $a, b \in \mathbb{Z}$  and  $f, g \in R$ , we have  $\phi(af + bg) = a\phi(f) + b\phi(g)$  and  $\phi(f)\Phi(g) = \phi(fg \bmod x^N + 1)$ .

We denote vectors by bold lowercase letters and use the infinity-norm  $\|\mathbf{v}\| := \|\mathbf{v}\|_\infty$ . For any  $f \in R$ , we define  $\|f\| = \|\phi(f)\|$ . Notice that  $\|fg\| \leq N\|f\|\|g\|$ . We denote matrices by bold capital letters and use the max-norm  $\|\mathbf{A}\| := \|\mathbf{A}\|_{\max} = \max\{|a_{i,j}| : a_{i,j} \text{ is an entry of } \mathbf{A}\}$ . If the entries of both  $\mathbf{A}$  and  $\mathbf{B}$  belong to  $R$ , then,  $\|\mathbf{A} \cdot \mathbf{B}\| \leq mN\|\mathbf{A}\| \cdot \|\mathbf{B}\|$ , where  $m$  is the number of rows of  $\mathbf{B}$ . If at least one of the matrices is integral, then  $\|\mathbf{A} \cdot \mathbf{B}\| \leq m\|\mathbf{A}\| \cdot \|\mathbf{B}\|$ .

Integer intervals are denoted with double brackets, e.g., an integer interval open on  $a$  and closed on  $b$  is  $\llbracket a, b \rrbracket = \mathbb{Z} \cap ]a, b]$ . The notation  $[x]_m$  means the only integer  $y$  in  $[-m/2, m/2[$  such that  $x = y \bmod m$ . When applied to vectors or matrices,  $[\cdot]_m$  is applied entry-wise, when applied to polynomials, it is applied to each coefficient. We define the column vector  $\mathbf{g} := (1, b, b^2, \dots, b^{\ell-1})^T$ . For any  $a \in \llbracket -b^\ell, b^\ell \rrbracket$ , let  $g^{-1}(a)$  be the signed base- $b$  decomposition of  $a$  such that the inner product  $g^{-1}(a)\mathbf{g}$  is equal to  $a$ . For a polynomial  $f$  with coefficients in  $\llbracket -b^\ell, b^\ell \rrbracket$ , we define  $g^{-1}(f) := \sum_{i=0}^{\deg(f)} g^{-1}(f_i)x^i$ . Thus,  $g^{-1}(f)\mathbf{g} = f$ .

## 2.2 Approximate-GCD problem

The Approximate Greatest Common Divisor problem was introduced in [HG01] and since then it has been used to construct several homomorphic encryption schemes [DGHV10, CCK<sup>+</sup>13, CS15]. The best known attacks against it run in exponential time [GGM16] and it is believed to be quantumly hard [BBL17]. Moreover, a variant of the problem in which the noise is sampled from a different distribution is equivalent to the LWE problem [CS15]. Now, we define this problem formally:

**Definition 1.** Let  $\rho, \eta, \gamma$ , and  $p$  be integers such that  $\gamma > \eta > \rho > 0$  and  $2^{\eta-1} \leq p \leq 2^\eta$ . The distribution  $\mathcal{D}_{\gamma, \rho}(p)$ , whose support is  $\llbracket 0, 2^\gamma \rrbracket$  is defined as

$$\mathcal{D}_{\gamma, \rho}(p) := \{\text{Sample } q \leftarrow \llbracket 0, 2^\gamma/p \rrbracket \text{ and } r \leftarrow \llbracket -2^\rho, 2^\rho \rrbracket : \text{Output } x := pq + r\}.$$

**Definition 2 (AGCD problem).** The  $(\rho, \eta, \gamma)$ -approximate-GCD problem is the problem of finding  $p$ , given many samples from  $\mathcal{D}_{\gamma, \rho}(p)$ .

The  $(\rho, \eta, \gamma)$ -decisional-approximate-GCD problem is the problem of distinguishing between  $\mathcal{D}_{\gamma, \rho}(p)$  and  $\mathcal{U}(\llbracket 0, 2^\gamma \rrbracket)$ .

## 2.3 Related work

**Fast bootstrapping using polynomial rings** In [DM15], the authors observed that in the polynomial ring  $\mathbb{Z}[x]/\langle x^N + 1 \rangle$  the element  $y := x^{2N/q}$  has order  $q$ . Thus, the multiplicative group  $\mathcal{G} := \langle y \rangle$  is isomorphic to  $\mathbb{Z}_q$ , in other words, we can map elements  $a_i$  from  $\mathbb{Z}_q$  to  $y^{a_i} \in \mathcal{G}$  and to add  $a_i + a_j \bmod q$  we compute  $y^{a_i} \cdot y^{a_j} \bmod x^N + 1 = y^{a_i + a_j \bmod q}$ . Additionally, representing  $\mathbb{Z}_q$  with  $\mathcal{G}$  is more efficient than using symmetric groups, as it was proposed in [ASP14], since it allows us to instantiate a GSW-like scheme based in the RLWE instead

of the LWE and to evaluate the decryption function of the base scheme by multiplying low-dimensional polynomial matrices instead of high-dimensional integral matrices.

Then, the authors of [DM15] propose a gate bootstrapping, i.e., they propose a simple base scheme that encrypts one bit and can evaluate one binary gate homomorphically, then it has to be bootstrapped. Thus, evaluating a binary circuit with this scheme requires that we perform the refreshing function after each gate. The binary gates are very efficient as they are performed with  $\Theta(n)$  simple additions modulo  $q$ , hence, the expensive part is to refresh the resulting ciphertext. The base scheme uses the LWE problem to encrypt a message  $m$  as  $\mathbf{c} := (\mathbf{a}, b := \mathbf{a}\mathbf{s} + e + mq/t \bmod q) \in \mathbb{Z}_q^{n+1}$ . The bootstrapping keys are GSW encryptions of the secret key  $\mathbf{s}$  essentially as follows:  $\mathfrak{R}_{i,j} = \text{GSW.Enc}(y^{-2^i \cdot s_j})$  for  $0 \leq i \leq \ell := \log(q)$  and  $1 \leq j \leq n$ . Then, given a ciphertext  $\mathbf{c} = (\mathbf{a}, b)$  to be refreshed, we write  $\mathbf{a} = (a_1, \dots, a_n)$ , decompose each  $a_j$  in base 2, obtaining  $(a_{0,j}, \dots, a_{\ell-1,j})$ , and the first step consists basically in using GSW's homomorphic product to compute  $b - \mathbf{a}\mathbf{s} = e + mq/t$  as follows:

$$\text{GSW.Enc}(y^b) \prod_{j=1}^n \prod_{\substack{0 \leq i < \ell: \\ a_{i,j} \neq 0}} \mathfrak{R}_{i,j} = \text{GSW.Enc}(y^{b - \sum_{j=1}^n a_{i,j} s_j \bmod q}).$$

The second step consists in transforming a GSW encryption of  $y^{e+mq/t}$  in a base scheme ciphertext encrypting  $m$ . Roughly speaking, this is done by taking the coefficient vector of one specific row of the GSW ciphertext and multiplying it by a fixed vector, then, applying a modulus- and a key-switching.

In [CGGI16], the authors noticed that instead of simply using the GSW homomorphic product, which consists in multiplying matrices of polynomials, we can perform the bootstrapping using a mixed product in which one operand is an LWE ciphertext (thus, a vector) and the another one is a GSW ciphertext (thus, a matrix), resulting then in a LWE ciphertext (again a vector). The authors called it an external product. This speeds up the bootstrapping since it replaces matrix-matrix products by vector-matrix multiplications.

Notice that in the context of AGCD-based schemes,  $q$  would be replaced by a secret  $p \in \Omega(2^\lambda)$  and we would need  $N \approx p$ , thus, the degree of the polynomials encrypted by the GSW-like scheme would be exponentially large. Moreover, since  $N$  would be public and  $2N \in p\mathbb{Z}$ , it would be possible to recover  $p$ .

**GSW-like schemes over the integers** In [BBL17], the authors use the AGCD problem to construct a GSW-like leveled homomorphic encryption scheme that encrypts a single bit  $m$  into a vector  $\mathbf{c} := p\mathbf{q} + \mathbf{r} + m\mathbf{g} \in \mathbb{Z}^\gamma$  where  $p\mathbf{q} + \mathbf{r} \leftarrow (\mathcal{D}_{\gamma,\rho}(p))^\gamma$  and  $\mathbf{g} = (2^0, 2^1, \dots, 2^{\gamma-1})$ . To perform homomorphic products, they define the operator  $\mathbf{G}^{-1}(\mathbf{c}) \in \{0,1\}^{\gamma \times \gamma}$  as a matrix such that each column  $j$  is  $g^{-1}(c_j)$ , that is, the binary decomposition of the  $j$ -th entry of  $\mathbf{c}$ . Notice that  $\mathbf{g}\mathbf{G}^{-1}(\mathbf{c}) = \mathbf{c}$ , thus, two ciphertexts  $\mathbf{c}_i := p\mathbf{q}_i + \mathbf{r}_i + m_i\mathbf{g}$  (for  $i = 1, 2$ ) are

multiplied homomorphically as

$$\begin{aligned}
\mathbf{c}_{mult} &:= \mathbf{c}_1 \mathbf{G}^{-1}(\mathbf{c}_2) \\
&= p \mathbf{q}_1 \mathbf{G}^{-1}(\mathbf{c}_2) + \mathbf{r}_1 \mathbf{G}^{-1}(\mathbf{c}_2) + m_1 \mathbf{g} \mathbf{G}^{-1}(\mathbf{c}_2) \\
&= p \underbrace{(\mathbf{q}_1 \mathbf{G}^{-1}(\mathbf{c}_2) + m_1 \mathbf{q}_2)}_{\mathbf{q}_{mult}} + \underbrace{(\mathbf{r}_1 \mathbf{G}^{-1}(\mathbf{c}_2) + m_1 \mathbf{r}_2)}_{\mathbf{r}_{mult}} + m_1 m_2 \mathbf{g}.
\end{aligned}$$

We see that the noise growth due to the homomorphic product is approximately additive, i.e.,  $\|\mathbf{r}_{mult}\| \leq \|\mathbf{r}_1 \mathbf{G}^{-1}(\mathbf{c}_2)\| + m_1 \|\mathbf{r}_2\| \leq \gamma \|\mathbf{r}_1\| + \|\mathbf{r}_2\|$ . However, this scheme is not practical. Their authors report that performing one single multiplication takes several seconds in a modern CPU. The main reason for this inefficiency is the huge ciphertext expansion, as it encrypts one bit into  $\gamma^2$  bits and, typically,  $\gamma$  is much bigger than  $\lambda$ .

Trying to amend this issue, in [Per20] it is proposed to expand the message space of [BBL17] so that instead of encrypting only bits, it is possible to encrypt vectors and matrices with non-binary entries. Furthermore, the ciphertexts are randomized with a hidden matrix  $\mathbf{K}$ , since, as it was observed in [CP19], all the attacks against the AGCD problem become much more expensive when the AGCD samples are multiplied by a random matrix and, thus, one can choose smaller parameters, in particular, one can decrease the size of  $\gamma$  and have better ciphertext expansion. The resulting scheme is a GSW-like leveled homomorphic scheme that can perform operations with matrices and vectors, in particular, it is possible to do homomorphic vector-matrix products. Notice that by representing elements of  $R$  with coefficient vectors and circulant matrices, we can use this scheme to operate homomorphically over  $R$ . In particular, we could, in principle, use it in a bootstrapping procedure à la [DM15]. However, by doing so, we would encrypt a degree- $N$  polynomial into a matrix ciphertext of dimension  $N\ell \times N$ , which would yield very large bootstrapping keys.

Hence, we go one step further and propose to randomize the AGCD problem with a random polynomial  $k(x)$  instead of a random matrix. Thereby we can encrypt a polynomial of degree  $N$  into an  $\ell$ -dimensional vector whose each entry is a degree- $N$  polynomial, gaining thus a factor  $N$ . We also define two types of ciphertexts and we provide an efficient homomorphic product between them. This corresponds to the vector-matrix product of [Per20] and to the external product of [CGGI16].

### 3 Randomized (Polynomial) AGCD problem

We start by extending the AGCD problem to a problem that is strictly equivalent, but that works on polynomials. Then, we propose to randomize this problem with a hidden polynomial  $k(x)$ , obtaining thus the underlying problem that will be used in our scheme.

**Definition 3 (Underlying distribution of PAGCD).** *Let  $N, \rho, \eta, \gamma$ , and  $p$  be integers such that  $\gamma > \eta > \rho > 0$  and  $p$  is an  $\eta$ -bit integer. The distribution*

$\mathcal{P}_{N,\gamma,\rho}(p)$ , whose support is  $\llbracket 0, 2^\gamma - 1 \rrbracket^N$ , is defined as

$$\mathcal{P}_{N,\gamma,\rho}(p) := \left\{ \text{Sample } c_0, \dots, c_{N-1} \leftarrow \mathcal{D}_{\gamma,\rho}(p) : \text{Output } c := \sum_{i=0}^{N-1} c_i x^i \right\}.$$

**Definition 4 (PAGCD).** *The  $(N, \rho, \eta, \gamma)$ -polynomial-approximate-GCD problem is the problem of finding  $p$ , given many samples from  $\mathcal{P}_{N,\gamma,\rho}(p)$ .*

*The  $(N, \rho, \eta, \gamma)$ -decisional-PAGCD problem is the problem of distinguishing between  $\mathcal{P}_{N,\gamma,\rho}(p)$  and  $\mathcal{U}(\llbracket 0, 2^\gamma \rrbracket^N)$ .*

Because each coefficient of each polynomial output by  $\mathcal{P}_{N,\gamma,\rho}(p)$  is an independent sample of  $\mathcal{D}_{\gamma,\rho}(p)$ , having  $N$  samples of the AGCD problem is the same as having one sample of the PAGCD problem, hence, it is clear that the PAGCD and the original AGCD problem are equivalent.

Now, aiming to choose smaller parameters and following the ideas of [CP19] and [Per20], we propose a randomized version of this problem, but instead of randomizing a vector of AGCD samples with a hidden matrix  $\mathbf{K}$ , we randomize a sample of  $\mathcal{P}_{N,\gamma,\rho}(p)$  with a hidden polynomial  $k$ , performing the operations in the ring  $R := \mathbb{Z}[x]/\langle x^N + 1 \rangle$ .

**Definition 5 (Underlying distribution of RAGCD).** *Let  $N, \rho, \eta, \gamma$ , and  $p$  be integers such that  $\gamma > \eta > \rho > 0$  and  $p$  is an  $\eta$ -bit integer. Let  $R := \mathbb{Z}[x]/\langle x^N + 1 \rangle$ ,  $x_0$  be a sample from  $\mathcal{D}_{\gamma,\rho}(p)$ , and  $k$  be a random invertible polynomial of  $R/x_0R$ . The distribution  $\mathcal{R}_{N,\gamma,\rho,x_0}(p, k)$ , whose support is  $R/x_0R$  is defined as*

$$\mathcal{R}_{N,\gamma,\rho,x_0}(p, k) := \{ \text{Sample } c \leftarrow \mathcal{P}_{N,\gamma,\rho}(p) : \text{Output } \tilde{c} := c \cdot k \in R/x_0R \}.$$

**Definition 6 (RAGCD).** *The  $(x_0, N, \rho, \eta, \gamma)$ -RAGCD problem is the problem of finding  $p$  and  $k$ , given  $x_0$  and many samples from  $\mathcal{R}_{N,\gamma,\rho,x_0}(p, k)$ .*

*The  $(x_0, N, \rho, \eta, \gamma)$ -decisional-RAGCD problem is the problem of distinguishing between  $\mathcal{R}_{N,\gamma,\rho,x_0}(p, k)$  and  $\mathcal{U}(R/x_0R)$ .*

Notice that we can define this problem using  $\mathbb{Z}[x]/\langle f \rangle$  for other polynomials  $f$ , but for our purposes,  $x^N + 1$  is enough. Moreover, given many instances of PAGCD problem, we can select one coefficient of any polynomial to be the scalar  $x_0$ , then sample a random invertible  $k$ , and multiply each PAGCD instance by  $k$  in  $R/x_0R$ , obtaining thus valid instances of the RAGCD problem. Thus, this problem cannot be easier than the PAGCD problem. Therefore, because the PAGCD and the original AGCD problem are equivalent, the RAGCD problem is no easier than the AGCD problem.

In the next lemma, we prove that if we fix  $x_0 \geq 2^{\gamma-1}$  and restrict the distribution  $\mathcal{R}_{N,\gamma,\rho,x_0}(p, k)$  so that it only randomizes polynomials with coefficients smaller than  $x_0$ , then we obtain a distribution that is indistinguishable from  $\mathcal{U}(R/x_0R)$  under the hardness of the decisional AGCD problem.



**Lemma 1.** *Let  $x_0$  be a sample of  $\mathcal{D}_{\gamma,\rho}(p)$  such that  $x_0 \geq 2^{\gamma-1}$ . Let  $\mathcal{D}_{<x_0}$  be the distribution obtained by rejecting samples of  $\mathcal{D}_{\gamma,\rho}(p)$  that are bigger than or equal to  $x_0$ . Let  $\mathcal{R}_{<x_0}$  be defined as  $\mathcal{R}_{N,\gamma,\rho,x_0}(p,k)$ , but randomizing polynomials with coefficient smaller than  $x_0$ , that is:*

$$\mathcal{R}_{<x_0} := \{ \text{Sample } c \leftarrow \mathcal{D}_{<x_0}^N : \text{Output the polynomial } \tilde{c} := c \cdot k \in R/x_0R \}.$$

*Then, distinguishing between  $\mathcal{R}_{<x_0}$  and  $\mathcal{U}(R/x_0R)$  is not easier than the decisional AGCD problem.*

*Proof.* Because  $x_0 \geq 2^{\gamma-1}$ , from Lemma 3 of (the full version of) [Per20], we know that the distributions  $\mathcal{D}_{<x_0}$  and  $\mathcal{U}(\mathbb{Z}_{x_0})$  are indistinguishable under the decisional AGCD assumption. Therefore, polynomials from  $R$  with coefficients sampled from  $\mathcal{D}_{<x_0}$  or from  $\mathcal{U}(\mathbb{Z}_{x_0})$  are also indistinguishable. But because  $k$  is invertible in  $R$  modulo  $x_0$ , multiplying by  $k$  does not change the distributions, that is,

$$\mathcal{D}_{<x_0}^N = \mathcal{D}_{<x_0}^N \bmod x_0 = \mathcal{D}_{<x_0}^N \cdot k \bmod x_0 = \mathcal{R}_{<x_0}$$

and

$$\mathcal{U}(\mathbb{Z}_{x_0}^N) = \mathcal{U}(\mathbb{Z}_{x_0}^N) \bmod x_0 = \mathcal{U}(\mathbb{Z}_{x_0}^N) \cdot k \bmod x_0 = \mathcal{U}(R/x_0R).$$

Hence,  $\mathcal{R}_{<x_0}$  and  $\mathcal{U}(R/x_0R)$  are indistinguishable.  $\square$

## 4 GSW-like AGCD-based Homomorphic Encryption

In this section, we present the GSW-like AGCD-based Homomorphic Encryption (GAHE) scheme that will be used to perform the bootstrapping. We start with a basic scheme that can encrypt a polynomial  $f \in R$  into a vector  $\mathbf{c} \in R^\ell$ . Then, by assuming circular security, we extend the definition of the scheme so that we also have scalar ciphertexts. Finally, we define a functional key-switching. For brevity and because in our main applications, the fast bootstrapping procedures, we only use the mixed homomorphic product, we omit the other homomorphic operations, like additions and “vector-vector” product, presenting them only in Appendix A. Furthermore, to ease the presentation, specially the noise-growth analysis, we keep the modulus  $x_0$  private. Hence, the homomorphic operations are performed on  $R$  instead of  $R/x_0R$ , which means that the bit length of the ciphertext grows. However, in our bootstrapping procedures, they grow only by a small factor that is independent of the multiplicative depth of the homomorphic evaluation.

- **GAHE.KeyGen**( $1^\lambda, N, t, b$ ): Choose the parameters  $\eta, \rho$ , and  $\gamma$ . Sample an  $\eta$ -bit random prime  $p$ . Sample  $x_0$  from  $p \cdot \mathcal{U}(\llbracket 1, 2^\gamma/p \rrbracket)$ , until  $x_0 \geq 2^{\gamma-1}$ . Then, sample  $k$  uniformly from  $R/x_0R$  until  $k^{-1}$  exists over  $R/x_0R$ . Define  $\ell_0 := \lceil \log_b(2^\gamma) \rceil$  and  $\ell := \lceil \ell_0 + \log_b(N) + 1 + \log_b(\ell_0 + \log_b(N) + 1) \rceil^1$ . The public parameters are **params** :=  $\{N, t, \ell, b, \eta, \gamma, \rho\}$  and secret key is **sk** :=  $(p, k, x_0)$ .

<sup>1</sup> If we were publishing  $x_0$ , then the homomorphic operations could be done modulo  $x_0$  and we could set  $\ell = \ell_0$ , without adding these extra logarithmic terms.

- **GAHE.EncVec**(sk,  $m$ ): Given a polynomial  $m \in R/tR$ , construct a vector  $\mathbf{x} := (p\mathbf{q} + \mathbf{r})k \in R^\ell$  by sampling each entry  $x_i$  independently from  $\mathcal{R}_{N,\gamma,\rho,x_0}(p, k)$ , then output the following vector  $\mathbf{c}$ :

$$\mathbf{c} := [\mathbf{x} + \mathbf{g} \cdot m]_{x_0} \in R^\ell.$$

- **GAHE.DecVec**(sk,  $\mathbf{c}$ ): Let  $\alpha := \lfloor p/t \rfloor$ . Compute  $c := \langle g^{-1}([\alpha k]_{x_0}), \mathbf{c} \rangle$  over  $R/x_0R$ . Then do  $c' := c \cdot k^{-1} \in R/x_0R$  and output

$$\left\lfloor \frac{t \cdot [c']_p}{p} \right\rfloor \bmod t.$$

#### 4.1 Assuming circular security to extend the scheme

In this section we show that, by assuming circular security, we can encrypt an element of  $R/tR$  into a single element of  $R$  instead of into a vector. We call a ciphertext produced by this new encryption method a scalar ciphertext and the ones produced by the encryption function defined before are vector ciphertexts. Moreover, we define the mixed homomorphic product between a vector and a scalar ciphertext. It is worth noticing that circular security is regarded as a weak assumption and has been used extensively in all types of homomorphic encryption schemes.

Thus, notice that by assuming circular security, we can use **GAHE.EncVec** to encrypt  $m \cdot k \cdot \lfloor p/t \rfloor$ , obtaining  $\mathbf{c} = (p\mathbf{q} + \mathbf{r})k + (m \cdot k \cdot \lfloor p/t \rfloor)\mathbf{g} = (p\mathbf{q} + \mathbf{r} + m \cdot \lfloor p/t \rfloor \cdot \mathbf{g})k$ . But then, because the first entry of  $\mathbf{g}$  is 1, we see that the first entry of  $\mathbf{c}$  has the following format:  $c_1 = (pq_1 + r_1 + m \cdot \lfloor p/t \rfloor)k \in R$ . Thus, we can extend our scheme with the following procedures:

- **GAHE.EncScalar**(sk,  $m$ ): Given a polynomial  $m \in R/tR$ , sample  $x := (pq + r)k \leftarrow \mathcal{R}_{N,\gamma,\rho,x_0}(p, k)$  and output

$$c := [x + m \cdot \lfloor p/t \rfloor \cdot k]_{x_0} \in R.$$

- **GAHE.DecScalar**(sk,  $c$ ): Output  $\left\lfloor \frac{t \cdot [c']_p}{p} \right\rfloor \bmod t$  where  $c' := c \cdot k^{-1} \in R/x_0R$ .
- **GAHE.MultMix**( $c, \mathbf{c}$ ): to perform a homomorphic mixed product, we decompose and multiply the scalar ciphertext  $c$  by the vector ciphertext  $\mathbf{c}$ , outputting the following inner product over  $R$ :  $c_{mult} := g^{-1}(c) \cdot \mathbf{c} \in R$ .

#### 4.2 Correctness of decryption

In this section we define the noise of a ciphertext and show the necessary conditions for the decryption functions to work.

**Definition 7 (Noise of scalar ciphertext).** Let  $c = (pq + r + \lfloor p/t \rfloor m)k$  be a scalar ciphertext encrypting a message  $m \in R/tR$ . We define the noise of  $c$  as  $\text{err}(c) := [(c \cdot k^{-1} - \lfloor p/t \rfloor m) \bmod x_0]_p$ . Notice that  $\text{err}(c)$  is exactly  $r$  if  $\|r\| < p/2$ .

**Definition 8 (Noise of vector ciphertext).** Let  $\mathbf{c} = (p\mathbf{q} + \mathbf{r})k + \mathbf{g}m$  be a vector encryption of  $m \in R/tR$ . We define the noise of  $\mathbf{c}$  as  $\text{err}(\mathbf{c}) := [(c - \mathbf{g}m) \cdot k^{-1} \bmod x_0]_p$ . Notice that  $\text{err}(\mathbf{c})$  is  $\mathbf{r}$  if  $\|\mathbf{r}\| < p/2$ .

**Lemma 2 (Upper bound on the noises).** Let  $c = (pq + r + \alpha m_1)k \in R$  be a scalar ciphertext and  $\mathbf{c} = (p\mathbf{q} + \mathbf{r})k + \mathbf{g}m_2 \in R^\ell$  be a vector ciphertext. Assuming that  $\|\text{err}(c)\|$  and  $\|\text{err}(\mathbf{c})\|$  are both smaller than  $p/2$ , it holds that  $\|\text{err}(c)\| = \|r\|$  and  $\|\text{err}(\mathbf{c})\| = \|\mathbf{r}\|$ . In particular, if  $c$  and  $\mathbf{c}$  are fresh ciphertexts, then  $\|\text{err}(c)\| < 2^\rho$  and  $\|\text{err}(\mathbf{c})\| < 2^\rho$ .

*Proof.* It is clear that  $\text{err}(c) = r$ , therefore,  $\|\text{err}(c)\| = \|r\|$ . Moreover, for fresh ciphertexts, we have  $\|r\| < 2^\rho$  because each coefficient of  $r$  is sampled uniformly from  $[-2^\rho, 2^\rho]$ . Basically the same argument applies to  $\text{err}(\mathbf{c})$ .  $\square$

Let's first analyze `GAHE.DecScalar`. Then, the correctness of `GAHE.DecVec` follows basically by the same argument.

**Lemma 3 (Correctness of scalar decryption).** Let  $c$  be a scalar encryption of  $m \in R/tR$ . If  $\|\text{err}(c)\| < \frac{p}{3t}$ , then `GAHE.DecScalar`( $\text{sk}, c$ ) outputs  $m$ .

*Proof.* Let  $c = (pq + r + \lfloor p/t \rfloor m)k$ . Consider the polynomial  $c' = c \cdot k^{-1} \in R/x_0R$  defined in `GAHE.DecScalar`. We can write it as  $c' = pq' + r + \lfloor p/t \rfloor m \in R$ . Then, when we perform the reduction modulo  $p$ , we obtain  $\bar{c} = [r + \lfloor p/t \rfloor m]_p = [\text{err}(c) + \lfloor p/t \rfloor m]_p = \text{err}(c) + \epsilon + mp/t - pu$  for some  $\epsilon, u \in R$  with  $\|\epsilon\| \leq 1/2$ .

Thus, in the next step of the decryption function, we have

$$\frac{t\bar{c}}{p} = \frac{t(\text{err}(c) + \epsilon)}{p} + m - ut.$$

But because  $\|\text{err}(c)\| < \frac{p}{2t}$ , we have  $\|t(\text{err}(c) + \epsilon)/p\| < 1/3 + \|t\epsilon/p\| < 1/2$ . Hence, since  $m - ut$  has integer coefficients, the rounding function outputs

$$\left\lfloor \frac{t\bar{c}}{p} \right\rfloor = \left\lfloor \frac{t(\text{err}(c) + \epsilon)}{p} \right\rfloor + m - ut = m - ut.$$

Therefore, the reduction modulo  $t$  indeed gives us  $m$ .  $\square$

**Lemma 4 (Sufficient conditions for correctness of vector decryption).**

Let  $\mathbf{c}$  be a vector encryption of  $m \in R/tR$ . If  $\|\text{err}(\mathbf{c})\| < \frac{p}{3N\ell b t}$ , then `GAHE.DecVec`( $\text{sk}, \mathbf{c}$ ) outputs  $m$ .

*Proof.* Let  $\alpha := \lfloor p/t \rfloor$ . Notice that `GAHE.DecVec`( $\text{sk}, \mathbf{c}$ ) can be rewritten as

1. Compute a scalar encryption of the same message  $m$ , i.e.,  $c := g^{-1}([\alpha k]_{x_0}) \cdot \mathbf{c}$ .
2. Output `GAHE.DecScalar`( $\text{sk}, c$ ).

But by definitions 7 and 8, we have

$$\text{err}(c) = [(c \cdot k^{-1} - \alpha m \bmod x_0)]_p = [g^{-1}([\alpha k]_{x_0}) \cdot \text{err}(\mathbf{c})]_p.$$

But  $\|g^{-1}([\alpha k]_{x_0}) \cdot \text{err}(\mathbf{c})\| \leq N\ell b \|\text{err}(\mathbf{c})\| < p/(3t)$ . Therefore, the output of `GAHE.DecScalar`( $\text{sk}, c$ ) is  $m$  by Lemma 3.  $\square$

### 4.3 Analysis of mixed homomorphic product

Let  $\mathbf{c}$  be a vector encryption of  $v$  and  $c$  be scalar encryption of  $s$ . Also, let  $\mathbf{y} := g^{-1}(c) \in R^\ell$ . In the definition of  $\text{GAHE.MultMix}(c, \mathbf{c})$  we have  $c_{mult} := \mathbf{y} \cdot \mathbf{c}$ , thus, the following holds:

$$\begin{aligned} c_{mult} &= (p\mathbf{y}\mathbf{q} + \mathbf{y}\mathbf{r})k + \mathbf{y}g v && \text{(By definition of } \mathbf{c} \text{)} \\ &= (p\mathbf{y}\mathbf{q} + \mathbf{y}\mathbf{r})k + cv && \text{(Because } \mathbf{y}g = c \text{)} \\ &= (p \underbrace{(\mathbf{y}\mathbf{q} + qv)}_{q_{mult}} + \underbrace{(\mathbf{y}\mathbf{r} + rv)}_{r_{mult}} + \lfloor p/t \rfloor sv)k && \text{(By definition of } c \text{)} \end{aligned}$$

Therefore, the mixed homomorphic product takes encryptions of  $s$  and  $v$  and produces  $c_{mult} = (pq_{mult} + r_{mult} + \lfloor p/t \rfloor sv)k \in R$ , which is a valid scalar encryption of the product of the messages, as expected.

As for the noise growth, we now show that a sequence of  $n$  mixed homomorphic products increases the noise just linearly in  $n$ .

**Lemma 5 (Noise growth of mixed products).** *Let  $n \in \mathbb{N}^*$ . For all  $i \in \llbracket 1, n \rrbracket$ , let  $\mathbf{c}_i$  be a vector encryption of  $m_i$ . Let also  $c_0$  be a scalar encryption of  $m_0$ . Assume that  $B$  is an upper bound to the norm of the products of plaintexts, i.e.,  $\|\prod_{i=j}^n m_i\| \leq B$  for  $0 \leq j \leq n$ . Finally, for  $1 \leq i \leq n$ , define  $c_i := \text{GAHE.MultMix}(c_{i-1}, \mathbf{c}_i) \in R$  (notice that  $c_i$  is a scalar encryption of  $\prod_{j=0}^i m_j$ ). Then,*

$$\|\text{err}(c_n)\| < NB \|\text{err}(c_0)\| + \sum_{i=1}^n N^2 B \ell b \|\text{err}(\mathbf{c}_i)\|. \quad (1)$$

In particular, if  $c_0$  and all the  $\mathbf{c}_i$ 's are fresh ciphertexts, then

$$\|\text{err}(c_n)\| < 2N^2 B \ell b n 2^\rho. \quad (2)$$

*Proof.* By the analysis done above, we know that the term  $r_i$  of  $c_i$  is  $g^{-1}(c_{i-1})\mathbf{r}_i + r_{i-1}m_i$ . Hence, the term  $r_n$  after  $n$  homomorphic products is

$$r_n = r_0 \prod_{i=1}^n m_i + \sum_{i=1}^n g^{-1}(c_{i-1})\mathbf{r}_i \left( \prod_{j=i+1}^n m_j \right) \in R.$$

Thus,

$$\begin{aligned} \|r_n\| &\leq N \|r_0\| \left\| \prod_{i=1}^n m_i \right\| + \sum_{i=1}^n N \ell \|g^{-1}(c_i)\| \left\| \mathbf{r}_i \left( \prod_{j=i+1}^n m_j \right) \right\| \\ &\leq NB \|r_0\| + \sum_{i=1}^n N^2 \ell b B \|\mathbf{r}_i\|. \end{aligned}$$

Therefore, Inequality 1 holds. By Lemma 2, if all the operands are fresh ciphertexts, then,  $\|r_0\| < 2^\rho$  and  $\|\mathbf{r}_i\| < 2^\rho$ , and the particular case also holds.  $\square$

#### 4.4 Functional Key-Switching

In this section we define a procedure that will play a main role in our bootstrapping methods, namely, a *functional* key-switching. Therewith we can change the keys and the dimension of the polynomial ring of a ciphertext and at the same time apply some function to the plaintext. That is to say, given two integers  $N_1$  and  $N_2$ , we define two polynomial rings  $R_i := \mathbb{Z}[x]/\langle x^{N_i} + 1 \rangle$ . Then, we can transform a scalar ciphertext  $c_1 \in R_1$  that encrypts a message  $m \in R_1/tR_1$  under key  $(p_1, k_1)$  into a ciphertext  $c_2$  encrypting  $\phi(m) \cdot \mathbf{u}$  under another key  $(p_2, k_2)$  for any  $\mathbf{u} \in R_2^{N_1}$ , where  $\phi(m) \in \mathbb{Z}^{N_1}$  is the coefficient vector of  $m$ .

Like the key-switching procedures of LWE-based schemes, our functional key-switching consists in two parts: firstly, we need both private keys to generate a functional key-switching key; then, using this key, we can publicly perform the transformation.

- **FuncKeySwGen**( $\text{sk}_1, \text{sk}_2, \text{params}, \mathbf{u}$ ): given  $\text{params} = (N_1, N_2, \tilde{b}, \tilde{\ell}, \gamma_2, \rho_2)$ , secret keys  $\text{sk}_i = (p_i, k_i) \in \mathbb{Z} \times R_i$ , and a vector  $\mathbf{u} \in R_2^{N_1}$ , proceed as follows:
  1. Define  $\mathbf{g}_{\tilde{b}} := (\tilde{b}^0, \dots, \tilde{b}^{\tilde{\ell}-1}) \in \mathbb{Z}^{\tilde{\ell} \times 1}$  and  $\mathbf{G} = \mathbf{I}_{N_1} \otimes \mathbf{g}_{\tilde{b}} \in \mathbb{Z}^{N_1 \tilde{\ell} \times N_1}$ .
  2. Let  $\mathbf{v} := \left\lfloor \frac{p_2}{p_1} \mathbf{G} \Phi(k_1^{-1}) \mathbf{u} \right\rfloor \in R_2^{N_1 \tilde{\ell}}$ , where  $p_2/p_1$  must be interpreted as a fraction in  $\mathbb{Q}$  and the inverse of  $k_1$  is computed on  $R_1/p_1R_1$ .
  3. Sample  $M$  from  $p_2 \cdot \mathcal{U}([0, 2^{\gamma_2}/p_2])$ .
  4. Sample  $\mathbf{y}$  from  $(\mathcal{P}_{N_2, \gamma_2, \rho_2}(p_2))^{N_1 \tilde{\ell}}$ .
  5. Output  $\text{swk} := [(\mathbf{y} + \mathbf{v}) \cdot k_2]_M$ . Notice that the output is of the form

$$\left( p_2 \mathbf{q} + \mathbf{r} + \left\lfloor \frac{p_2}{p_1} \mathbf{G} \Phi(k_1^{-1}) \mathbf{u} \right\rfloor \right) \cdot k_2 \in R_2^{N_1 \tilde{\ell}}.$$

- **FuncKeySw**( $c_1, \text{swk}$ ): Given a scalar ciphertext  $c_1 \in R_1$  and a functional key-switching key  $\text{swk} \in R_2^{N_1 \tilde{\ell}}$ , define  $\mathbf{z} := \phi(c_1) \in \mathbb{Z}^{N_1}$ , decompose each entry of  $\mathbf{z}$  in base  $\tilde{b}$  as  $\mathbf{w} := (g^{-1}(z_1), \dots, g^{-1}(z_{N_1})) \in \mathbb{Z}^{N_1 \tilde{\ell}}$ , and output  $c_2 := \mathbf{w} \cdot \text{swk} \in R_2$ .

**Lemma 6 (Correctness of functional key switching).**

Let  $\mathbf{u} \in R_2^{N_1}$ ,  $\text{sk}_i := (p_i, k_i) \in \mathbb{Z} \times R_i$ , and  $\text{params} := (N_1, N_2, \tilde{b}, \tilde{\ell}, \gamma_2, \rho_2)$ . Let also  $\text{swk} := \text{FuncKeySwGen}(\text{sk}_1, \text{sk}_2, \text{params}, \mathbf{u})$ . Then, for any  $c_1 \in R_1$  encrypting  $m \in R_1/tR_1$  under key  $\text{sk}_1$ , it holds that  $c_2 := \text{FuncKeySw}(c_1, \text{swk})$  is a valid encryption of  $\phi(m) \cdot \mathbf{u} \in R_2$  under key  $\text{sk}_2$ , if  $\|c_1\| < \tilde{b}^{\tilde{\ell}}$ . Moreover, the noise term of  $c_2$  is bounded as follows:

$$\|\text{err}(c_2)\| \leq \tilde{\ell} N_1 \tilde{b} 2^{\rho_2} + 2^{\eta_2 - \eta_1 + 2} N_1 \|\mathbf{u}\| \|\text{err}(c_1)\|$$

where  $\eta_i$  is the bit length of  $p_i$ .

*Proof.* Let  $c_1 = (p_1 q_1 + r_1 + \alpha_1 m) k_1 \in R_1$ , where  $\alpha_1 := \lfloor p_1/t \rfloor$ . Notice that  $\mathbf{w}$  defined in **FuncKeySw** satisfies  $\mathbf{w} \mathbf{G} = \phi(c_1)$  because  $\|c_1\| < \tilde{b}^{\tilde{\ell}}$ . Moreover,

$\phi(c_1)\Phi(k^{-1}) = p_1\mathbf{q}_1 + \phi(r_1) + \alpha_1\phi(m)$ . Therefore, the output of FuncKeySwT is

$$\begin{aligned}
c_2 &= (p_2\mathbf{w}\mathbf{q} + \mathbf{w}\mathbf{r} + \mathbf{w}\epsilon + \frac{p_2}{p_1}\mathbf{w}\mathbf{G}\Phi(k^{-1})\mathbf{u}) \cdot k_2 && \text{(For some } \|\epsilon\| \leq 1/2) \\
&= (p_2\mathbf{w}\mathbf{q} + \mathbf{w}(\mathbf{r} + \epsilon) + \frac{p_2}{p_1}(p_1\mathbf{q}_1 + \phi(r_1) + \alpha_1\phi(m))\mathbf{u})k_2 \\
&= (p_2q_2 + \mathbf{w}(\mathbf{r} + \epsilon) + \frac{p_2}{p_1}(\phi(r_1) + \alpha_1\phi(m))\mathbf{u}) \cdot k_2 && \text{(For } q_2 := \mathbf{w}\mathbf{q} + \mathbf{q}_1\mathbf{u}) \\
&= (p_2q_2 + \mathbf{w}(\mathbf{r} + \epsilon) + \frac{p_2}{p_1}(\phi(r_1) + \epsilon\phi(m))\mathbf{u} + \frac{p_2}{t}\phi(m)\mathbf{u}) \cdot k_2 && \text{(For some } \epsilon \in R_2)
\end{aligned}$$

Therefore,  $c_2$  is indeed an encryption of  $\phi(m)\mathbf{u}$  with respect to the key  $\mathbf{sk}_2$ , that is,  $c_2 = (p_2q_2 + r_2 + p_2\phi(m)\mathbf{u}/t)k_2 \in R_2$  with  $\text{err}(c_2) = r_2 = \mathbf{w}(\mathbf{r} + \epsilon) + \frac{p_2}{p_1}(\phi(r_1) + \epsilon\phi(m))\mathbf{u}$ . Furthermore,

$$\begin{aligned}
\|\text{err}(c_2)\| &\leq \|\mathbf{w}\mathbf{r}\| + \|\mathbf{w}\epsilon\| + \left\| \frac{p_2}{p_1}(\phi(r_1) + \epsilon\phi(m))\mathbf{u} \right\| \\
&\leq \tilde{\ell}N_1\tilde{b}\|\mathbf{r}\| + \tilde{\ell}N_1\tilde{b}/2 + 2^{\eta_2 - \eta_1 + 1}N_1\|\mathbf{u}\|(\|\text{err}(c_1)\| + t/2) \\
&\leq \tilde{\ell}N_1\tilde{b}2^{\rho_2} + 2^{\eta_2 - \eta_1 + 2}N_1\|\mathbf{u}\|\|\text{err}(c_1)\|. \quad \square
\end{aligned}$$

It turns out that this procedure is very general. For example, if we set  $\mathbf{u} = (1, x, \dots, x^{N_1})$ , then,  $\phi(m)\mathbf{u} = \sum_{i=0}^{N_1} m_i x^i = m$ , therefore, by using such  $\mathbf{u}$ , our functional key-switching works as an ordinary key-switching, outputting an encryption of the same message  $m$  but in the ring  $R_2$  and under the key  $\mathbf{sk}_2$ . By setting  $\mathbf{u} = (1, z, \dots, z^{N_1})$  for any  $z \in \mathbb{Z}$ , we obtain an encryption of  $\phi(m)\mathbf{u} = m(z)$ , i.e., the evaluation of  $m$  at the point  $z$ . Also notice that when  $N_i = 1$ , we have  $R_i \simeq \mathbb{Z}$ , thus, not only our procedure is well defined for  $N_i = 1$ , but it also switches the underlying problem from the AGCD to the RAGCD problem or vice versa.

#### 4.5 Semantic security

A vector ciphertext encrypts a message  $m \in R$  by adding  $x_i + m \bmod x_0$  for values  $x_i$  sampled from  $\mathcal{R}_{N,\gamma,\rho,x_0}(p,k)$ . Thus, if we assume that it is hard to distinguish between  $\mathcal{U}(R/x_0R)$  and  $\mathcal{R}_{N,\gamma,\rho,x_0}(p,k)$ , it is clear that it will also be hard to distinguish vector ciphertexts from uniform values, because  $x_i$  and  $x_i + m \bmod x_0$  follow the same distribution. Moreover, our scalar encryption can be viewed as a particular case of vector encryption if we assume circular security. Therefore, we have the following result, which can be made formal by a standard proof via a sequence of hybrids.

**Lemma 7.** *Under the decisional-RAGCD assumption and the circular-security assumption, encryptions of any pair of polynomials are computationally indistinguishable.*

## 5 Single-gate bootstrapping

In this section, we show how to use our scheme to bootstrap a simple AGCD-based “single-gate” homomorphic encryption scheme as it was done in the RLWE-based fast bootstrapping methods of [DM15,CGGI16].

### 5.1 Base scheme

Consider the following simple AGCD-based scheme that will be used as the base scheme, that is, as the scheme that will be bootstrapped. As it is done in [DM15,CGGI16], this base scheme is a leveled scheme with two levels only, thus, fresh ciphertexts are at level-1, we can evaluate one homomorphic binary gate by performing some simple additions, obtaining a ciphertext at level-2, and then we have to refresh the ciphertext to reduce the noise and to go back to level 1. Since all the binary gates can be written as compositions of logical NAND gates, to keep the presentation simple, we just present this binary gate. Furthermore, to avoid confusion, we represent the parameters of the base scheme with an overscore. For instance, the secret key of the base scheme is a prime  $\bar{p}$  of bit length  $\bar{\eta}$ , while GAHE’s secret key has an  $\eta$ -bit prime  $p$ .

- **HE.ParamGen**( $\lambda$ ): Choose  $\bar{p} = \lambda$ ,  $\bar{\eta} = \bar{p} + \beta$  for some small constant  $\beta$ , and  $\bar{\gamma} = \max(2\bar{\eta}, \lceil \beta^2 \lambda / \log(\lambda) \rceil)$ . Output **params** :=  $(\bar{\gamma}, \bar{\eta}, \bar{p}, \lambda)$ .
- **HE.KeyGen**(**params**): Sample a random prime  $\bar{p}$  from  $\llbracket 2^{\bar{\eta}-1}, 2^{\bar{\eta}} \rrbracket$  and  $\bar{p}q_{\text{ek}} + r_{\text{ek}} \leftarrow \mathcal{D}_{\bar{\gamma}, \bar{p}}(\bar{p})$ . Define the evaluation key as  $\bar{\text{ek}} := \bar{p}q_{\text{ek}} + r_{\text{ek}} + \lfloor 5\bar{p}/8 \rfloor$  and the secret key as  $\bar{\text{sk}} := \bar{p}$ .
- **HE.Enc**( $\bar{\text{sk}}, m, L$ ): To encrypt a bit  $m$ , sample  $\bar{p}q + r \leftarrow \mathcal{D}_{\bar{\gamma}, \bar{p}}(\bar{p})$  and output the level- $L$  ciphertext  $c = \bar{p}q + r + \lfloor L\bar{p}/4 \rfloor m$ .
- **HE.Dec**( $\bar{\text{sk}}, c$ ): To decrypt a level-1  $c$ , compute  $c' := \lfloor c/\bar{p} \rfloor$ , then output  $\left\lfloor \left\lfloor \frac{4c'}{\bar{p}} \right\rfloor \right\rfloor_2$ .
- **HE.Nand**( $c_1, c_2, \bar{\text{ek}}$ ): Let  $c_0$  and  $c_1$  be level-1 ciphertexts encrypting  $m_1$  and  $m_2$ , respectively. Output  $c := \bar{\text{ek}} - c_1 - c_2$ .

The function **HE.ParamGen** chooses the parameters in a way that guarantees the correctness of **HE.Dec**. Namely, because  $|r| < 2^{\bar{p}}$ , we have  $|r + \lfloor \bar{p}/4 \rfloor m| < \bar{p}/2$ , therefore,  $c' = \lfloor c/\bar{p} \rfloor = r + \lfloor \bar{p}/4 \rfloor m$  in  $\mathbb{Z}$ . And since  $\lfloor \bar{p}/4 \rfloor > 2^{\bar{p}+1} > 2|r|$ , we have  $\lfloor 4r/\bar{p} \rfloor = 0$ , then, the output is  $\lfloor 4c'/\bar{p} \rfloor = \lfloor 4r/\bar{p} \rfloor + m = m$ .

Our NAND gate is the same of [DM15,CGGI16], thus it outputs

$$c = \bar{p} \left( \underbrace{(q_{\text{ek}} - q_1 - q_2)}_{q_{\text{nand}}} + \underbrace{r_{\text{ek}} - r_1 - r_2}_{r_{\text{nand}}} \pm \frac{\bar{p}}{8} + \lfloor \bar{p}/2 \rfloor (1 - m_1 m_2) \right)$$

which is a level-2 encryption of  $\text{NAND}(m_1, m_2)$  with noise  $|r_{\text{nand}}| < 3 \cdot 2^{\bar{p}} + \bar{p}/8$ .

By standard techniques [DGHV10,CS15,BBL17] one can prove that this base scheme is CPA-secure if the AGCD problem is computationally hard. Moreover, the parameters chosen in **HE.ParamGen** provide a concrete security of  $\lambda$  bits.

## 5.2 Generating the bootstrapping keys

To generate the key material used to bootstrap, we need to fix a base  $B \geq 2$  in which we decompose the ciphertexts of the base scheme when they are refreshed. Then, we define  $L := \lceil \log_B(2^{\bar{\gamma}}) \rceil$ , which is the number of words needed to decompose the given ciphertexts. Moreover, the number of homomorphic mixed products that we perform during the refresh procedure is  $\Theta(L)$ , thus, there is a time-memory trade-off, as the amount of memory increases in general when we increase  $B$ , but at the same time,  $L$  decreases.

The bootstrapping procedure consists in two main steps: in the first one, we use the GAHE scheme to homomorphically multiply a given ciphertext  $\bar{c}$  by  $N/\bar{p}$ , obtaining a GAHE's scalar ciphertext  $c$ ; in the second step, we transform  $c$  in a valid base scheme ciphertext  $c'$ . To perform the first step, we would like to encrypt values of the form  $y^{gB^i N/\bar{p}}$ , where  $y := x^2$ , but the exponent would not be integer, thus, we encrypt  $y^{\lfloor gB^i N/\bar{p} \rfloor}$ , that is, we define  $\mathfrak{K}_{g,i} := \text{GAHE.EncVec}\left(y^{\lfloor gB^i N/\bar{p} \rfloor}\right)$  for  $1 \leq g < B$  and  $0 \leq i < L$ . In addition, we also encrypt an integer  $\delta$  that is added to the result obtained in the first step, so that the final result is contained in the interval  $\llbracket 0, N-1 \rrbracket$ . Thus, we define  $\mathfrak{K}_\delta := \text{GAHE.EncScalar}(y^\delta) \in R$ .

Notice that  $\mathfrak{K}_\delta$  is a scalar ciphertext, while  $\mathfrak{K}_{g,i}$  are vector ciphertexts. During the refresh procedure, we use the mixed homomorphic product to multiply them. Hence, at the end of the first step, we have a scalar ciphertext  $c = (pq+r+\alpha y^e)k$  for some  $e \in \llbracket 0, N-1 \rrbracket$  whose value depends on the message  $m$ . Then, to extract  $m$ , we define a test vector  $\mathbf{u} \in \{0, 1\}^N$ , such that  $\phi(y^e) \cdot \mathbf{u} = 1 - 2m$  and use our functional key-switching to transform  $c$  into a ciphertext that encrypts  $\phi(y^e) \cdot \mathbf{u}$  under the base scheme key  $\bar{\text{sk}}$ . Thus, we also append the following key to the bootstrapping keys:

$$\text{ek} := \text{FuncKeySwGen}(\text{sk} := (p, k), \bar{\text{sk}} := (\bar{p}, 1), \mathbf{u}).$$

In Algorithm 1 we present in detail the procedure to generate the bootstrapping key  $\text{bk}$ .

## 5.3 Refreshing a ciphertext

The goal of the bootstrapping is to take a level-2 ciphertext  $c = \bar{p}q + r + \lfloor \bar{p}/2 \rfloor m \in \mathbb{Z}$  whose noise term satisfies  $|r| < 3 \cdot 2^{\bar{p}} + \bar{p}/8$  and to output a level-1 ciphertext  $c' = \bar{p}q' + r' + \lfloor \bar{p}/4 \rfloor m$  with  $|r'| < 2^{\bar{p}}$ . The refreshing procedure is shown thoroughly in Algorithm 2 and it consists in two main steps: in the first one, we decompose  $c$  in the base  $B$  obtaining  $(c_0, c_1, \dots, c_{L-1})$ , then we use the bootstrapping key and GAHE's mixed homomorphic multiplication to obtain a scalar encryption of  $y^e$ , where  $y := x^2$  and

$$y^e = y^\delta \cdot \prod_{i=0}^{\ell-1} y^{\lfloor c_i B^i N/\bar{p} \rfloor} = y^{\delta + cN/\bar{p} + \epsilon \bmod N} = y^{\delta + rN/\bar{p} + mN/2 + \epsilon}$$



---

**Algorithm 1: GENBOOTSTRAPKEYS**


---

**Input:** Decomposition base  $B$ , secret key  $\bar{p}$  of the base scheme  
**Output:** Bootstrapping key  $\mathbf{bk}$

- 1  $y \leftarrow x^2$
- 2  $L \leftarrow \lceil \bar{\gamma} \cdot \log_B(2) \rceil$
- 3 **for**  $g = 1$  **until**  $B - 1$  **do**
- 4     **for**  $i = 0$  **until**  $L - 1$  **do**
- 5          $\mathfrak{K}_{g,i} \leftarrow \text{GAHE.EncVec}(y^{\lfloor gB^i N/\bar{p} \rfloor})$
- 6          $\epsilon_{g,i} \leftarrow gB^i N/\bar{p} - \lfloor gB^i N/\bar{p} \rfloor$
- 7 Choose  $\Delta \in \mathbb{N}$  such that  $|\sum_{i=0}^{L-1} \epsilon_{g,i}| < \Delta$  for any  $(g_0, \dots, g_{L-1})$ , e.g.,  $\Delta = L/2$
- 8  $\delta \leftarrow \lceil \Delta + (3 \cdot 2^{\bar{p}} + \bar{p}/8)N/\bar{p} \rceil$
- 9  $\mathfrak{K}_\delta \leftarrow \text{GAHE.EncScalar}(y^\delta)$  using  $\alpha := \lfloor p/8 \rfloor$
- 10  $\gamma_{\text{ek}} \leftarrow \lfloor \bar{\gamma} - \log(\ell Nb) \rfloor$  and  $\rho_{\text{ek}} \leftarrow \lfloor \bar{p} - \log(\ell Nb) - 2 \rfloor$
- 11  $\text{params} \leftarrow (N, 1, b, \ell, \gamma_{\text{ek}}, \rho_{\text{ek}})$
- 12  $\mathbf{u} \leftarrow (1, 1, \dots, 1) \in \mathbb{Z}^N$
- 13  $\text{ek} \leftarrow \text{FuncKeySwTGen}(\text{sk}_1 := (\bar{p}, 1), \text{sk}_2 := (p, k), \text{params}, \mathbf{u}) \in \mathbb{Z}^{N\ell}$
- 14  $\mathfrak{K}_8 \leftarrow \mathcal{D}_{\bar{\gamma}, \bar{p}-1}(\bar{p}) + \lfloor \bar{p}/8 \rfloor$
- 15 **return**  $\mathbf{bk} := \left( \text{ek}, \mathfrak{K}_\delta, \mathfrak{K}_8, \{\mathfrak{K}_{g,i}\}_{\substack{1 \leq g < B \\ 0 \leq i < L}} \right)$

---

for some small value  $\epsilon$ .

Then, notice that  $\phi(y^e) = \phi(x^{2^e})$ , thus, if  $0 \leq e < N/2$ , then, the only non-zero entry of  $\phi(y^e)$  is 1, otherwise, it is  $-1$ . But as we show in Lemma 8, we have  $0 \leq e < N/2$  if  $m = 0$  and  $N/2 \leq e \leq N - 1$  if  $m = 1$ , therefore, the vector  $\mathbf{u} := (1, \dots, 1) \in \mathbb{Z}^N$  satisfies  $\phi(y^e) \cdot \mathbf{u} = 1 - 2m$ . Thus, because we used  $\mathbf{u}$  to generate  $\text{ek}$ , when we apply the functional key-switching in the second step of Algorithm 2, we switch from GAHE to the base scheme and we obtain an encryption of  $\phi(x^e) \cdot \mathbf{u} = 1 - 2m$ , that is, we obtain  $\tilde{c} = \bar{p}\tilde{q} + \tilde{r} + (1 - 2m) \cdot \lfloor p/8 \rfloor$ . Therefore, subtracting  $\tilde{c}$  from  $\mathfrak{K}_8$  yields a valid level-one base scheme encryption of  $m$ , that is,  $c' := \mathfrak{K}_8 - \tilde{c} = \bar{p}q' + r' + m \lfloor p/4 \rfloor$ .

In which follows, we prove the correctness of the refreshing procedure.

**Lemma 8.** *Let  $c$  be a level-2 encryption of  $m \in \{0, 1\}$  with noise bounded by  $3 \cdot 2^{\bar{p}} + \bar{p}/8$ . Let  $N \geq 4\delta$  where  $\delta$  is the integer defined in Algorithm 1. Then, the ciphertext  $z \in R$  obtained at the end of the main loop of Algorithm 2 is an encryption of  $x^{2^e}$  where  $e \in \llbracket 0, N - 1 \rrbracket$ . Moreover,  $m = 0 \iff 0 \leq e < N/2$  and  $m = 1 \iff N/2 \leq e < N$ .*

*Proof.* Let  $y := x^2$ . We initialize  $z$  with a scalar encryption of  $y^\delta$  and at each iteration  $i$ , we add  $\lfloor c_i b^i N/\bar{p} \rfloor$  to the exponent, thus, because  $y$  has order  $N$  in  $R$ , it is clear that at the end of the loop  $z$  encrypts  $y^e$  where  $e = \delta + \sum_{i=0}^{L-1} \lfloor c_i B^i N/\bar{p} \rfloor \pmod N$ .

---

**Algorithm 2: REFRESH**


---

**Input:** Level-2 ciphertext  $c$  of the base scheme, bootstrapping key  $\text{bk}$   
**Output:** Level-1 ciphertext  $c$  of the base scheme

- 1 Let  $(c_0, c_1, \dots, c_{L-1})$  be a decomposition of  $c$  in base  $B$ .
- 2  $z \leftarrow \mathfrak{R}_\delta$
- 3 **for**  $i = 0$  **until**  $L - 1$  **do**
- 4     **if**  $c_i > 0$  **then**
- 5          $z \leftarrow \text{GAHE.MultMix}(z, \mathfrak{R}_{c_i, i})$
- $\triangleright$  **Second step: extract the message.**
- 6  $\tilde{c} \leftarrow \text{FuncKeySwT}(z, \text{ek})$
- 7  $c' \leftarrow \mathfrak{R}_8 - \tilde{c}$
- 8 **return**  $c'$

---

Now, let  $\epsilon_i := \lfloor c_i B^i N / \bar{p} \rfloor - c_i B^i N / \bar{p}$  and  $\epsilon := \sum_{i=0}^{L-1} \epsilon_i$ . Then,

$$e = \delta + \epsilon + (N/\bar{p}) \cdot \sum_{i=0}^{L-1} c_i B^i = \delta + \epsilon + cN/\bar{p} = \delta + \epsilon + rN/\bar{p} + mN/2 \pmod{N}$$

Notice that  $|\epsilon + rN/\bar{p}| < \Delta + (3 \cdot 2^{\bar{p}} + \bar{p}/8)N/\bar{p} \leq \delta$ . Also, because  $N \geq 4\delta$  by hypothesis, we have  $2\delta \leq N/2$ . Thus, we can see that

$$\delta + \epsilon + rN/\bar{p} + mN/2 < 2\delta + \frac{N}{2} \leq N.$$

Similarly,  $\delta + \epsilon + rN/\bar{p} + mN/2 \geq \delta - \Delta - \frac{(3 \cdot 2^{\bar{p}} + \bar{p}/8)N}{\bar{p}} \geq \delta - \delta = 0$ . Therefore, we conclude that  $0 \leq e < N$ . But because  $e$  is integer, we have  $0 \leq e \leq N - 1$ , as expected.

Thereby,  $e = \delta + \epsilon + rN/\bar{p} + mN/2$  over  $\mathbb{Z}$ , without the reduction modulo  $N$ . Thus,

$$\begin{cases} m = 0 \implies e = \delta + \epsilon + rN/\bar{p} < 2\delta \leq N/2 \\ m = 1 \implies e = \delta + \epsilon + rN/\bar{p} + N/2 \geq N/2 \end{cases} \quad \square$$

**Lemma 9.** *Let  $m \in \{0, 1\}$  and  $\delta \in \mathbb{N}^*$  as defined in Algorithm 1. Let  $e \in \llbracket 0, N - 1 \rrbracket$  such that  $m = 0 \iff 0 \leq e < N/2$ . Let  $z$  be a scalar encryption of  $x^{2^e}$  with  $\alpha = \lfloor p/8 \rfloor$  and noise bounded by some value  $B_z$ . Then, when given  $z$  as input, the second step of Algorithm 2 outputs a base scheme level-1 encryption of  $m$  with noise bounded by  $2^{\bar{p}-1} + NB_z 2^{\bar{\eta}-\eta+2}$ .*

*Proof.* We know that  $z = (pq_z + r_z + \lfloor p/8 \rfloor x^{2^e})k$  and that  $\text{ek}$  is a functional switching key from  $\text{sk} = (p, k)$  to  $\bar{\text{sk}} = (\bar{p}, 1)$  with respect to  $\mathbf{u} = (1, \dots, 1)$ . Then, by Lemma 6, the output of  $\text{FuncKeySwT}(z, \text{ek})$  is

$$\tilde{c} = \bar{p}\tilde{q} + \tilde{r} + \left\lfloor \frac{\bar{p}}{8} \right\rfloor \phi(x^{2^e})\mathbf{u} \in \mathbb{Z},$$

where  $|\text{err}(\tilde{c})| \leq 2^{\bar{p}-2} + NB_z 2^{\bar{\eta}-\eta+2}$  and  $\phi(x^{2^e})\mathbf{u} = 1 - 2m$ .

Notice that  $\lfloor p/8 \rfloor - (1 - 2m) \lfloor p/8 \rfloor = 2m \lfloor p/8 \rfloor = m \lfloor p/4 \rfloor + \epsilon$ , therefore, the output  $c' := \mathfrak{K}_8 - \tilde{c}$  is indeed of the form  $\bar{p}q' + r' + m \lfloor p/4 \rfloor$ , which a valid base scheme level-1 encryption of  $m$ . Moreover,  $\text{err}(c') \leq \text{err}(\mathfrak{K}_8) + \text{err}(\tilde{r}) \leq 2^{\bar{\rho}-1} + NB_z 2^{\bar{\eta}-\eta+2}$   $\square$

**Theorem 1 (Correctness of bootstrapping).** *Let  $c$  be a level-2 encryption of  $m \in \{0,1\}$  with noise bounded by  $3 \cdot 2^{\bar{\rho}} + \bar{p}/8$ . Let  $N \geq 4\delta$  where  $\delta$  is the integer defined in Algorithm 1. Let  $\bar{\rho} \geq \rho + \bar{\eta} - \eta + \log(N^3 \ell b L) + 4$ . Then, the refresh procedure, Algorithm 2, outputs a valid base scheme level-1 encryption of  $m$  with noise smaller than  $2^{\bar{\rho}}$ .*

*Proof.* By Lemma 8, we know that the ciphertext  $z$  produced at the end of the first step of REFRESH is of the form  $(pq_z + r_z + \lfloor p/8 \rfloor x^{2^e})k$  for some  $e \in \llbracket 0, N-1 \rrbracket$  such that  $m = 0 \iff 0 \leq e < N/2$ . Therefore, by Lemma 9, the output of REFRESH is a base scheme level-1 ciphertext  $c' = \bar{p}q' + r' + \lfloor \bar{p}/4 \rfloor m$  with  $|r'| < 2^{\bar{\rho}-1} + N \cdot \|\text{err}(z)\| \cdot 2^{\bar{\eta}-\eta+2}$ . But because  $z$  is computed with a sequence of  $L$  mixed homomorphic products, by Lemma 5, we have  $\|\text{err}(z)\| < 2N^2 \ell b L 2^{\rho}$ , therefore,  $|r'| < 2^{\bar{\rho}-1} + 2^{\rho+\bar{\eta}-\eta+\log(N^3 \ell b L)+3} \leq 2^{\bar{\rho}-1} + 2^{\bar{\rho}-1} = 2^{\bar{\rho}}$ .  $\square$

#### 5.4 Truncating ciphertexts to speed up refreshing

When we encrypt a message  $m$  with the base scheme, we multiply it by a constant  $\alpha$  that is bigger than the noise. This has the effect of shifting the message so that it is encrypted “between” the noise and the key. It has already been noticed [Bra12,CS15] that when we use noisy encryption schemes that encrypt messages in this way, we can discard the least significant bits of the ciphertexts, at the expense of increasing the noise, to use less memory to represent encrypted messages. But in our case, because the main loop of Algorithm 2 ignores bits equal to zero, we can also save some homomorphic multiplications and speed up the bootstrapping. Moreover, we do not need to generate bootstrapping keys  $\mathfrak{K}_{g,i}$ 's for these truncated bits.

In detail, given a base scheme ciphertext  $c = \bar{p}q + r + \bar{\alpha}m$ , we can set the first  $\mu$  bits to zero, for example, by subtracting  $s := c \bmod 2^\mu$  from  $c$ . Notice that we obtain then  $c' := c - s = \bar{p}q + (r - s) + \bar{\alpha}m$ , that is a valid encryption of  $m$ , but with a new noise term potentially bigger, satisfying  $\text{err}(c') \leq \text{err}(c) + 2^\mu$ . Then, when we decompose  $c'$  in base  $B$  in the refreshing procedure, the first  $\mu_B := \lfloor \mu \cdot \log_B(2) \rfloor$  words are ignored and the keys  $\mathfrak{K}_{g,i}$  for  $0 \leq i < \mu_B$  are never used. Thus, we reduce the number of mixed homomorphic products from  $L$  to  $L - \mu_B$  and the number of keys from  $(B - 1)L$  to  $(B - 1)(L - \mu_B)$ .

## 6 Multi-value bootstrapping

In this section, we extend and adapt our single-gate bootstrapping to solve the more general problem of bootstrapping a scheme whose message space is  $\mathbb{Z}_t$  instead of  $\{0,1\}$  and that can perform more than one homomorphic operation before refreshing. Moreover, we show how to change the test vector so

that instead of simply outputting  $m$ , the bootstrapping outputs encryptions of  $f_1(m), \dots, f_n(m)$ , for arbitrary functions  $f_i : \mathbb{Z}_t \rightarrow \mathbb{Z}_t$ . As before, the parameters of the base scheme are denoted with an overscore. In particular, we have now two polynomial rings:  $R := \mathbb{Z}[x]\langle x^N + 1 \rangle$ , used by GAHE, and  $\bar{R} := \mathbb{Z}[\bar{x}]\langle \bar{x}^{\bar{N}} + 1 \rangle$ , used by the base scheme. We assume that ciphertexts of the base scheme are of the form  $(\bar{p}q + r + \bar{\alpha}m)\bar{k} \in \bar{R}$ , for  $\bar{\alpha} := \lfloor \bar{p}/(2t) \rfloor$  and a degree- $\bar{N}$  polynomial  $\bar{k}$ . Some existing AGCD-based FHE schemes, as [CS15], encrypt a message  $m$  in a ciphertext of the form  $\bar{p}q + r + \bar{\alpha}m \in \mathbb{Z}$ , thus, they can be viewed as a particular case when  $\bar{N} = 1$ . Alternatively, one can adapt [CS15] to use the polynomial ring  $\bar{R}$  instead of  $\mathbb{Z}$  and to randomize the ciphertexts so that they have the format we are proposing. Other schemes, like [DGHV10], use  $c = pq + tr + m$  as the ciphertext format, but it is sufficient to apply a simple transformation on these ciphertexts to obtain our expected format, i.e., just homomorphically multiply  $c$  by the integer  $(p - \lfloor p \rfloor_t)/t$ . For completeness, we discuss this simple transformation in Appendix B.

## 6.1 Generating the multi-value bootstrapping keys

To perform the refreshing procedure we have to multiply the ciphertexts by  $\bar{k}^{-1}$  and perform the reduction modulo  $\bar{p}$ . As in the single-gate case, instead of performing the bootstrapping on  $\mathbb{Z}_{\bar{p}}$ , we work on  $\mathbb{Z}_N$ . And rather than multiplying by  $\bar{k}^{-1}$ , we multiply by the first column of the circulant matrix of  $\bar{k}^{-1}$ , i.e., if we define  $\mathbf{k}_1 := \text{column}_1(\Phi(\bar{k}^{-1}))$ , then,  $\Phi(\bar{k})\mathbf{k}_1 = \mathbf{e}_1 := (1, 0, \dots, 0) \in \{1, 0\}^{\bar{N}}$ , hence, given  $c = (\bar{p}q + r + \bar{\alpha}m)\bar{k} \in \bar{R}$ , it holds that

$$\phi(c)\mathbf{k}_1 = \phi(\bar{p}q + r + \bar{\alpha}m)\Phi(\bar{k})\mathbf{k}_1 = \phi(\bar{p}q + r + \bar{\alpha}m)\mathbf{e}_1 = \bar{p}q' + r_0 + \bar{\alpha}m \in \mathbb{Z}.$$

Therefore, one way of extending our single-gate bootstrapping is by multiplying the exponents encrypted in the bootstrapping keys by the entries of  $\mathbf{k}_1$ , that is, by defining  $\mathfrak{R}_{g,i,j} := \text{GAHE.EncVec}\left(x^{\lfloor gB^i k_{1,j} N / \bar{p} \rfloor}\right)$  for  $1 \leq g < B$ ,  $1 \leq j \leq \bar{N}$  and  $0 \leq i < L$ . We would also need to add one extra loop in the refreshing procedure, because now each entry  $j$  of  $\mathbf{a} := \phi(c)$  would be decomposed in base  $B$ , and we would compute  $\mathfrak{R}_\delta \prod_{j=1}^{\bar{N}} \prod_{i=0}^{L-1} \mathfrak{R}_{a_i,j,i,j}$ .

However the number of bootstrapping keys would be  $\bar{N}(B-1)L$  in lieu of  $(B-1)L$ , which could result in huge memory requirements, and the number of homomorphic mixed multiplications in the first step would increase from  $\Theta(L)$  to  $\Theta(\bar{N}L)$ , which could result in a considerable slowdown.

Thus, rather than inserting  $\mathbf{k}_1$  into the keys  $\mathfrak{R}_{g,i}$ , we define a “remove  $\bar{k}$  key”,  $\text{rk}$ , which is a functional key-switching key from the RAGCD to the AGCD problem with respect to the vector  $\mathbf{e}_1 := (1, 0, \dots, 0) \in \{0, 1\}^{\bar{N}}$ . Because the base scheme ciphertext  $c \in R$  encrypts an integer  $m$ , it is clear that  $\phi(m)\mathbf{e}_1 = m$ , thus, applying  $\text{rk}$  to  $c$  produces an integer encryption of  $m$  and we can then continue the bootstrapping as in the single-gate case.

Therefore, we generate the bootstrapping keys as follows:

1. Choose  $\eta_{rk}, \rho_{rk} = \Omega(\lambda)$  and  $\gamma_{rk} = \Omega(\lambda(\eta_{rk} - \rho_{rk})^2 / \log(\lambda))$ . Sample a  $\eta_{rk}$ -bit prime  $p_{rk}$ . Let  $\text{params} := (\bar{N}, 1, 2, \bar{\gamma}, \gamma_{rk}, \rho_{rk})$ ,  $\text{sk}_{rk} := (p_{rk}, 1)$ , and  $\bar{\text{sk}} := (\bar{p}, \bar{k})$ . Then, define

$$\text{rk} := \text{FuncKeySwGen}(\bar{\text{sk}}, \text{sk}_{rk}, \text{params}, \mathbf{e}_1).$$

2. Let  $L := \lceil (\log(\bar{N}\bar{\gamma}) + \gamma_{rk}) \log_B(2) \rceil$  instead of  $L = \lceil \bar{\gamma} \log_B(2) \rceil$  and generate  $\mathfrak{R}_{g,i}$ 's as in Algorithm 1, but using  $p_{rk}$ , i.e.,  $\mathfrak{R}_{g,i} := \text{GAHE.EncVec}(x^{2 \cdot \lfloor gB^i N/p_{rk} \rfloor})$ .
3. Let  $\beta$  be an upper bound to the noise of the ciphertexts that will be refreshed, define  $\delta := \lceil \Delta + \beta N/p_{rk} \rceil$ , and  $\mathfrak{R}_\delta := \text{GAHE.DecScalar}(x^{2\delta})$ , using  $\alpha := \lfloor p/(2t) \rfloor$ .
4. Finally, instead of defining a single extraction key  $\text{ek}$ , we define one key for each chosen function. Namely, we assume that  $N \geq 4t\delta$  and that the bootstrapping procedure also receives a list of functions  $f_i : \mathbb{Z}_t \rightarrow \mathbb{Z}_t$ , for  $1 \leq i \leq n$ . Then, we define the parameters  $\gamma_{ek}$  and  $\rho_{ek}$  as in Algorithm 1, and  $\text{params} := (N, \bar{N}, b, \ell, \gamma_{ek}, \rho_{ek})$ . Finally,

$$\text{ek}_i := \text{FuncKeySwGen}(\text{sk}, \bar{\text{sk}}, \text{params}, \mathbf{u}_i)$$

where  $\mathbf{u}_i \in \mathbb{Z}_t^N$  is a test vector defined as follows: for  $0 \leq m \leq t-1$ , all the entries  $\mathbf{u}_i[2 \cdot \lfloor Nm/(2t) \rfloor], \dots, \mathbf{u}_i[4\delta + 2 \cdot \lfloor Nm/(2t) \rfloor]$  are equal to  $f_i(m)$ . The remaining entries are zero. Expressly:

$$\mathbf{u}_i = \underbrace{(f_i(0), \dots, f_i(0))}_{4\delta \text{ entries}}, 0, \dots, 0, \underbrace{(f_i(1), \dots, f_i(1))}_{4\delta \text{ entries}}, 0, \dots, 0, \underbrace{(f_i(t-1), \dots, f_i(t-1))}_{4\delta \text{ entries}}, 0, \dots, 0$$

Notice that if  $Nm/(2t) < e < 2\delta + Nm/(2t)$ , then  $\phi(x^{2e}) \cdot \mathbf{u}_i = f_i(m)$ , thus, at the end of the bootstrapping, when we apply the functional key-switching to an encryption of  $\phi(x^{2e})$  using  $\text{ek}_i$ , we obtain an encryption of the desired value, i.e.,  $f_i(m)$ . We also remark that, because the test vectors  $\mathbf{u}_i$ 's are encrypted, the functions  $f_i$ 's are never disclosed, hence, we can dispatch the computation to a third party without revealing the functions that are being evaluated. The bootstrapping key is then defined as  $\text{bk} := \left( \text{rk}, \{\text{ek}_i\}_{1 \leq i \leq n}, \mathfrak{R}_\delta, \{\mathfrak{R}_{g,i}\}_{\substack{1 \leq g < B \\ 0 \leq i < L}} \right)$ .

## 6.2 The multi-value refreshing procedure

Refreshing a ciphertext  $\bar{c} = (\bar{p}\bar{q} + \bar{r} + \bar{\alpha}m)\bar{k} \in \bar{R}$  is similar to the single-gate case. Basically, we first transform  $\bar{c}$  in an integer ciphertext  $c$ , then, we proceed as in Algorithm 2 and compute a scalar encryption of  $x^{2e}$ . Then, instead of extracting the message one time, we extract  $f_i(m)$  for each  $f_i$  encoded in  $\text{ek}_i$ . We show it in detail in Algorithm 3.

It is easy to see that the bit length of the integer ciphertext  $c$  produced in the beginning of Algorithm 3 is bounded by  $\log(\bar{N}\bar{\gamma}) + \gamma_{rk}$ . Thus, because we use  $L = \lceil (\log(\bar{N}\bar{\gamma}) + \gamma_{rk}) \log_B(2) \rceil$  in the bootstrapping key generation, we

---

**Algorithm 3: MULTI-VALUE REFRESH**

---

**Input:** Base scheme ciphertext  $\bar{c} \in \bar{R}$ , bootstrapping key  $\text{bk}$   
**Output:** Level-1 ciphertext  $c$  of the base scheme

- 1  $c \leftarrow \text{FuncKeySwT}(\bar{c}, \text{rk}) \in \mathbb{Z}$   
▷ Now, the same main loop of Algorithm 2
- 2 Let  $(c_0, c_1, \dots, c_{L-1})$  be a decomposition of  $c$  in base  $B$ .
- 3  $z \leftarrow \mathfrak{K}_\delta$
- 4 **for**  $i = 0$  **until**  $L - 1$  **do**
- 5     **if**  $c_i > 0$  **then**
- 6          $z \leftarrow \text{GAHE.MultMix}(z, \mathfrak{K}_{c_i, i})$
- ▷ Then extract each  $f_i(m)$
- 7 **for**  $i = 1$  **until**  $n$  **do**
- 8      $c'_i \leftarrow \text{FuncKeySwT}(z, \text{ek}_i)$
- 9 **return**  $c'_1, \dots, c'_n$

---

can proceed as in the single-gate case, decomposing  $c$  in base  $B$  and obtaining a GAHE's scalar ciphertext  $z = (pq_z + r_z + \alpha x^{2e})k \in R$ , where  $e = \delta + \epsilon + cN/p_{\text{rk}} \bmod N = \delta + \epsilon + rN/p_{\text{rk}} + mN/(2t)$ . Because we chose  $\delta > |\epsilon + rN/p_{\text{rk}}|$ , we have,  $mN/(2t) < e < 2\delta + mN/(2t)$ . We want these intervals to be disjoint, thus, we need  $2\delta + mN/(2t) \leq (m + 1)N/(2t)$ , which is satisfied if we choose  $N \geq 4t\delta$ . With this choice, we have then  $\phi(x^{2e})\mathbf{u}_i = f_i(m)$ , which means that the last step extracts the correct messages.

Similarly to Lemma 9, assuming that the noise of  $z$  is bounded by  $B_z$ , the final noise of each  $c'_i$  is  $\|r'_i\| < 2^{\bar{\rho}-1} + B_z 2^{\log(Nt) + \bar{\eta} - \eta + 1}$ .

Then, as in Theorem 1, we have  $B_z = 2N^2 \ell b L 2^\rho$ , hence, if we choose  $\bar{\rho} \geq \rho + \log(2N^3 \ell b L t) + \bar{\eta} - \eta + 2$ , we have

$$\|r'_i\| < 2^{\bar{\rho}-1} + 2^{\rho + \log(2N^3 \ell b L t) + \bar{\eta} - \eta + 1} \leq 2^{\bar{\rho}-1} + 2^{\bar{\rho}-1} = 2^{\bar{\rho}}$$

in other words, the final noise after bootstrapping is smaller than the initial noise.

## 7 Practical results

In this section we show how to choose the parameters for our scheme, we present the running times and memory usage of our bootstrapping procedures, and we compare our results with previous works. Our proof-of-concept was implemented in C++ using the Number Theory Library<sup>2</sup> (NTL) . We ran the experiments on a single core of a processor Intel Core i5-8600K 3.60GHz, of a machine with 32GB of RAM memory. We stress that our implementation is not optimized and the running times that we present below, although being enough to show that our techniques are practical, can certainly be improved.

<sup>2</sup> <https://www.shoup.net/ntl/>

## 7.1 Cryptanalysis and parameter selection

Firstly we recall the definitions of the parameters used in our GAHE scheme:

- $N$ : we work over the cyclotomic ring  $R := \mathbb{Z}[x]/\langle x^N + 1 \rangle$ ;
- $\eta$ : we sample the secret prime  $p$  uniformly from  $\llbracket 2^{\eta-1}, 2^\eta \rrbracket$ ;
- $\rho$ : during encryption, we sample the noise terms uniformly from  $\llbracket -2^\rho, 2^\rho \rrbracket$ ;
- $\gamma$ : the private modulus  $x_0$  satisfies  $2^{\gamma-1} \leq x_0 < 2^\gamma$
- $t$ : the message space is  $R/tR = \mathbb{Z}_t[x]/\langle x^N + 1 \rangle$ ;
- $b$ : base in which we perform the decomposition  $g^{-1}$ ;
- $\ell$ : number of words used in  $g^{-1}$ . Vector ciphertexts belong to  $R^\ell$ .

In [CP19] and [Per20], the authors analyzed a randomized version of the AGCD problem in which the AGCD samples are arranged as vectors and multiplied by a hidden random matrix  $\mathbf{K} \in \mathbb{Z}^{N \times N}$ . Namely, for fixed  $p$  and  $\mathbf{K}$ , an attacker has access to many vectors  $\tilde{\mathbf{v}}_i = \mathbf{v}_i \mathbf{K} \in \mathbb{Z}^N$ , where  $\mathbf{v}_i = p\mathbf{q}_i + \mathbf{r}_i \in \mathbb{Z}^N$ . In our case, an attacker has access to polynomials  $\tilde{c}_i := c_i \cdot k = (pq_i + r_i)k \in R$  output from  $\mathcal{R}_{N,\gamma,\rho,x_0}(p,k)$ . But, denoting by  $\mathbf{K} \in \mathbb{Z}^{N \times N}$  the circulant matrix of  $k$ , we can write  $\phi(\tilde{c}_i) = \phi(c_i)\mathbf{K} = (p\phi(q_i) + \phi(r_i))\mathbf{K} \in \mathbb{Z}^N$  which can be viewed as the randomized AGCD problem of [CP19,Per20], but with a structured matrix  $\mathbf{K}$  instead of a completely random. Hence, we use the cryptanalysis done in [CP19,Per20] without taking advantage of the structure of the circulant matrix of  $k$ . The parallel with the RLWE problem is worthy of note: the attacks against the RLWE problem are adapted from the cryptanalysis of the LWE problem, since the RLWE can be seen as a structured version of the LWE. Also, in practice, such structure is ignored because there is no known way of exploiting it.

Thus, to guarantee the security, we must set  $\gamma \in \Omega\left(\frac{\lambda(\eta-\rho)^2}{N \log \lambda}\right)$  to rule out orthogonal lattice attacks and  $\rho \in \Omega(\lambda/N)$  to avoid GCD attacks. If we let  $L$  be the maximum multiplicative depth to be evaluated, then, the correctness of the decryption functions imposes the following constraint: By lemmas 3 and 4, decryption works if the final noise is smaller than  $p/(3t)$ . Thus, we can use  $2^\eta/(6t) < p/(3t)$  as an acceptable bound to the noise. By Lemma 5, the final noise is upper bounded by  $2N^2BL\ell b2^\rho$ , thus, we need  $2N^2BL\ell b2^\rho \leq 2^\eta/(6t)$ , or, equivalently,

$$\eta \geq \rho + \log(tN^2BL\ell b) + \log(12).$$

In sections 7.2 and 7.3 we propose concrete parameters sufficient to evaluate our bootstrapping methods.

**Dimension reduction:** We notice that if one wants to work over another ring of the form  $\mathbb{Z}[x]/\langle f \rangle$ , then  $f$  has to be carefully chosen, because non-trivial factors of  $f$  can be used to reduce the dimension of the RAGCD problem. Namely, over  $\mathbb{Z}[x]$ , the RAGCD instances can be written as  $c = (pq+r)k - uf$ , thus, if  $g$  divides  $f$  in  $\mathbb{Z}[x]$ , then, we can reduce  $c$  modulo  $g$  obtaining  $c' = (p[q]_g + [r]_g) \cdot [k]_g - vg$ , which is then a new RAGCD instance over the “smaller” ring  $\mathbb{Z}[x]/\langle g \rangle$ , because the degree of  $g$  is less than the degree of  $f$ .

That is the reason for us to choose  $f = x^N + 1$  with  $N$  being a power of two, since in this case,  $f$  is a cyclotomic polynomial, therefore, irreducible over  $\mathbb{Z}$ .

## 7.2 Practical results of single-gate bootstrapping

In this section we present our practical results for the bootstrapping procedure described in Section 5. Our experiment consisted in encrypting two random bits  $m_1$  and  $m_2$  into  $c_1$  and  $c_2$ , computing  $c := \text{HE.Nand}(c_1, c_2)$ , then refreshing  $c$ . Because the homomorphic NAND gate is performed with three simple integer additions, its running time is negligible and we only measured the refreshing step. To show the time-memory trade-off explicitly, we used several values for the base  $B$ . For the base scheme, we fixed  $\beta = 5$  and generated the other parameters as described in the function `HE.KeyGen` presented in Section 5.1. We also truncated  $\mu = \bar{\rho} - 5$  bits of the ciphertexts, as explained in Section 5.4. We recall that when we generate `bk`, we must choose an upper-bound  $\Delta$  for the sum of the rounding errors  $\epsilon_{g,i}$ 's. Because  $|\epsilon_{g,i}| \leq 1/2$ , it is clear that we can choose  $\Delta = L/2$ , however, this bound is not realistic, thus, we use  $\Delta = (L - \lambda \log_B(2))/6$ . Hence, we fixed  $\lambda = \eta = 100$  and, for each  $B$ , we defined  $L$  and the parameters of the GAHE scheme as presented in Table 1.

Our running times and memory requirements are comparable with those of [DM15] and around 10 times larger than those of [CGGI16]. A full comparison is presented in Table 1. We stress that this comparison is to be taken with care, because we used a 3.6 GHz processor while they used a 3.0 GHz one. On the other hand, while we used NTL's polynomial multiplication as a black box, they used very optimized Fast Fourier Transform (FFT) libraries to perform polynomial multiplications and they precomputed the FFTs of the bootstrapping keys, thus, in the main loop of the refreshing procedure, they just need to apply the FFT to one of the operands. If we had precomputed the FFTs, then, at each iteration of our bootstrapping we would need to compute only  $\ell$  FFTs (on each entry of  $g^{-1}(\mathfrak{K}_\delta) \in R^\ell$ ), thus, the total number of FFTs would decrease from around  $2L\ell$  to  $L\ell$ . Therefore, our running times can surely be improved.

## 7.3 Practical results of multi-value bootstrapping

In [CIM19] the authors propose some techniques to perform the fast bootstrapping procedures of [DM15] and [CGGI16] on non-binary message spaces. Then, they implement a multi-value bootstrapping that takes an encryption of a 6-bit message  $m$  and outputs an encryption of  $f(m)$  for an arbitrary function from  $\{0, 1\}^6$  to  $\{0, 1\}^6$ . Thus, to compare our results with theirs, we use  $t = 2^6$ . In this experiment, we fix a random function  $f$  from  $\mathbb{Z}_t$  to  $\mathbb{Z}_t$  and we generate the bootstrapping keys. Then we sample a random message  $m$  from  $\mathbb{Z}_t$ , we encrypt it as  $(\bar{p}q + r + \lfloor \bar{p}/(2t) \rfloor m)k$ , and we executed the bootstrapping procedure obtaining a ciphertext encrypting  $f(m)$ . As in [CIM19], we can apply several additional functions to the same message in a single bootstrapping almost for free, that is, outputting encryptions of  $f_1(m), \dots, f_n(m)$  for  $n > 1$  costs almost the same as outputting one single ciphertext, since the main loop of the bootstrapping



Table 1: We show the practical results of our single-gate bootstrapping for several sets of parameters. The two last rows show [DM15] and [CGGI16], which used only one fixed set of parameters. We show the running times they reported on a 3.0 GHz processor and also these timings multiplied by  $\frac{3}{3.6}$  to make the comparison with our results more sensible. The security level is  $\lambda = 100$  for the three schemes and we always used  $\gamma = 200$  for our GAHE scheme.

	$\log B$	$N$	$\rho$	$L$	$\log b$	Size bk	Refreshing
Ours	5	256	51	76	26	115 MB	0.57 s
	7	128	65	54	14	283 MB	0.28 s
	9	128	65	42	14	889 MB	0.22 s
[DM15]	-					1.3 GB	$0.69 \text{ s} (\times \frac{3}{3.6} = 0.57 \text{ s})$
[CGGI16]	-					52 MB	$0.05 \text{ s} (\times \frac{3}{3.6} = 0.04 \text{ s})$

Table 2: The first three rows show, with respect to  $B$ , the parameters used in GAHE and the practical results of our multi-value bootstrapping running on a 3.6 GHz processor. The last row shows the results reported in [CIM19] on a 3.5 GHz processor.

	$\log B$	$L$	$N$	$\rho$	$\log b$	$\ell$	Size bk	Refreshing
Ours	9	27	2048	2	56	6	2.6 GB	7.74 s
	11	22	1024	2	57	6	4.4 GB	1.38 s
[CIM19]	-						$\approx 8 \text{ GB}$	1.57 s

procedure is executed only one time and for each additional function we just have to compute one inner product  $\mathbf{w} \cdot \mathbf{e}k_i$  over  $\bar{R}$ .

To achieve 100 bits of security, we fixed the following parameters:

- *Base scheme*:  $\bar{N} = 8$ ,  $\bar{\eta} = 100$ ,  $\bar{\rho} = 85$ ,  $\bar{\gamma} = 423$ .
- *Key rk*:  $\eta_{rk} = 104$ ,  $\rho_{rk} = 88$ ,  $\gamma_{rk} = 231$ .
- *Decomposition*: for each base  $B$ , we set  $L = \lceil \log_B(241) \rceil$ .
- *Key  $\mathfrak{K}_\delta$* :  $\Delta = (L - \log_B(2^{100}))/6$  and  $\delta = \lceil 2\Delta \rceil$ .

Moreover, after applying the key rk to obtain an integer ciphertext  $c$ , we set  $\mu := \lambda - 7$  bits of  $c$  to zero, as discussed in Section 5.4, thus, we did not generate keys  $\mathfrak{K}_{g,i}$ 's for  $0 \leq i < \mu_B := \lfloor \mu \log_B(2) \rfloor$ . To instantiate GAHE, we used  $\eta = 100$  and  $\gamma = 200$ . The other parameters depend on  $B$  and are shown in Table 2 along with the sizes of our bootstrapping keys and the running times. The parameter  $N$  was chosen as the smaller power of two that is bigger than  $4t\delta$ . We also included the practical results of [CIM19] for comparison. Using around 4 GB of key material, which is approximately half of the key size used in [CIM19], our running times are already faster than those of [CIM19]. We remark that truncating  $\mu$  bits allowed us to save about 35 % of the memory

requirements, as the sizes of  $\text{bk}$  would have been 4.24 GB and 6.91 GB, instead of the values presented in Table 2 if we had not ignored the least significant bits of the ciphertexts.

## 8 Conclusion

In this work, we showed that bootstrapping procedures for FHE schemes over the integers are not inherently slower than for (R)LWE-based schemes. It would be interesting to find an efficient way of performing our multi-value bootstrapping on a base scheme that encrypts a polynomial  $m$  instead of an integer. If  $m$  were a binary polynomial of low degree, we could evaluate  $m$  at 2 (using our functional key-switching) to obtain an integer encryption, then, perform one single multi-value bootstrapping, but with one extraction key  $\text{ek}_i$  for each bit of  $m(2)$ . However, the general case remains as an open problem. Another natural question that arises is how to perform similar fast bootstrapping procedures to AGCD-based schemes that use the Chinese Remainder Theorem to pack several messages into one ciphertext, since that could significantly improve the amortized cost of bootstrapping.

## References

- [ASP14] Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In *Advances in Cryptology – CRYPTO 2014*, pages 297–314, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- [BBL17] Daniel Benarroch, Zvika Brakerski, and Tancrede Lepoint. FHE over the Integers: Decomposed and Batched in the Post-Quantum Regime. In *Public-Key Cryptography – PKC 2017*, pages 271–301, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
- [BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, ITCS '12*, pages 309–325, New York, NY, USA, 2012. ACM.
- [Bra12] Zvika Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology – CRYPTO 2012*, pages 868–886, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [BV11] Z. Brakerski and V. Vaikuntanathan. Efficient Fully Homomorphic Encryption from (Standard) LWE. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 97–106, Oct 2011.
- [CCK<sup>+</sup>13] Jung Hee Cheon, Jean-Sébastien Coron, Jinsu Kim, Moon Sung Lee, Tancrede Lepoint, Mehdi Tibouchi, and Aaram Yun. Batch fully homomorphic encryption over the integers. In *Advances in Cryptology – EURO-CRYPT 2013*, pages 315–335, Berlin, Heidelberg, 2013. Springer.
- [CGGI16] Iliaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology – ASIACRYPT 2016*, pages 3–33, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

- [CIM19] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New Techniques for Multi-value Input Homomorphic Evaluation and Applications. In *Topics in Cryptology – CT-RSA 2019*, pages 106–126, Cham, 2019. Springer International Publishing.
- [CNT12] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *Advances in Cryptology – EUROCRYPT 2012*, pages 446–464, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [CP19] Jean-Sebastien Coron and Hilder V. L. Pereira. On Kilian’s Randomization of Multilinear Map Encodings. In *ASIACRYPT 2019*, 2019. <https://eprint.iacr.org/2018/1129>.
- [CS15] Jung Hee Cheon and Damien Stehlé. Fully Homomorphic Encryption over the Integers Revisited. In *EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Sofia, Bulgaria, April 2015.
- [DGHV10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully Homomorphic Encryption over the Integers. In *Advances in Cryptology – EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*. Springer, 2010.
- [DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640, Berlin, Heidelberg, 2015. Springer.
- [Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. [crypto.stanford.edu/craig](http://crypto.stanford.edu/craig).
- [GGM16] Steven D. Galbraith, Shishay W. Gebregiyorgis, and Sean Murphy. Algorithms for the approximate common divisor problem. *LMS Journal of Computation and Mathematics*, 19(A):58–72, 2016.
- [GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *Proceedings of the 31st Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT’12, pages 465–482, Berlin, Heidelberg, 2012. Springer-Verlag.
- [GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology – CRYPTO 2013*, pages 75–92, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [HG01] Nick Howgrave-Graham. Approximate integer common divisors. In *Cryptography and Lattices*, pages 51–66, Berlin, Heidelberg, 2001. Springer.
- [Per20] Hilder Vitor Lima Pereira. Efficient AGCD-based homomorphic encryption for matrix and vector arithmetic. Cryptology ePrint Archive, Report 2020/491, 2020. <https://eprint.iacr.org/2020/491>.

## Appendix

### A Other homomorphic operations

- $\text{GAHE.AddVec}(\mathbf{c}_1, \mathbf{c}_2)$ : to homomorphically add two ciphertexts, just add them entry-wise:  $\mathbf{c}_{add} := \mathbf{c}_1 + \mathbf{c}_2 \in R^\ell$ .
- $\text{GAHE.MultVec}(\mathbf{c}_1, \mathbf{c}_2)$ : to perform a homomorphic product, apply  $g^{-1}$  to each entry of  $\mathbf{c}_1$  obtaining a  $\ell \times \ell$  matrix of polynomials, i.e.,  $\mathbf{A} := (g^{-1}(c_{1,1}) \dots g^{-1}(c_{1,\ell}))$ , then perform a vector-matrix product over  $R$ :  $\mathbf{c}_{mult} := \mathbf{c}_2 \cdot \mathbf{A} \in R^\ell$ .
- $\text{GAHE.AddScalar}(c_1, c_2)$ : to perform a homomorphic addition, just add the ciphertexts:  $c_{add} := c_1 + c_2 \in R$ .
- $\text{GAHE.AddPlaintext}(\mathbf{c}_1, h)$  and  $\text{GAHE.MultPlaintext}(\mathbf{c}_1, h)$ : to add a plaintext  $h$ , output  $\mathbf{c}_1 + \mathbf{g} \cdot h$ . To multiply, simply multiply each entry of  $\mathbf{c}_1$  by  $h$  in  $R$ , i.e., output  $h \cdot \mathbf{c}_1 \in R^\ell$ .

#### A.1 Correctness of homomorphic operations

The mixed homomorphic product was analyzed in Section 4.3. We now show that the other homomorphic operations are also correct. For  $i \in \{1, 2\}$ , let  $\mathbf{c}_i$  be a vector encryption of  $v_i$  and  $c_i$  be a scalar encryption of  $s_i$ . Thus, we have  $\mathbf{c}_i = (p\mathbf{q}_i + \mathbf{r}_i)k + \mathbf{g}v_i$  and  $c_i = (pq_i + r_i + \alpha s_i)k$ .

Hence, it is easy to see that the homomorphic additions produce valid ciphertexts, i.e.,

$$\begin{aligned} - c_1 + c_2 &= (p(q_1 + q_2) + (r_1 + r_2) + \alpha(s_1 + s_2))k \in R. \\ - \mathbf{c}_1 + \mathbf{c}_2 &= (p(\mathbf{q}_1 + \mathbf{q}_2) + (\mathbf{r}_1 + \mathbf{r}_2))k + \mathbf{g}(v_1 + v_2) \in R^\ell. \end{aligned}$$

To see that the homomorphic product of two vector ciphertexts is correct, notice that we decompose one of the operands, say,  $\mathbf{c}_1$ , as  $\mathbf{A} = (g^{-1}(c_{1,1}) \dots g^{-1}(c_{1,\ell})) \in R^{\ell \times \ell}$ , and when we multiply  $\mathbf{A}$  by  $\mathbf{g}$ , we obtain again  $\mathbf{c}_1$ , i.e.,  $\mathbf{g} \cdot \mathbf{A} = \mathbf{c}_1$ . Hence, we have the following:

$$\begin{aligned} \mathbf{c}_{mult} &= \mathbf{c}_2 \cdot \mathbf{A} \\ &= (p\mathbf{q}_2\mathbf{A} + \mathbf{r}_2\mathbf{A})k + \mathbf{g}\mathbf{A}v_2 \\ &= (p\mathbf{q}_2\mathbf{A} + \mathbf{r}_2\mathbf{A})k + ((p\mathbf{q}_1 + \mathbf{r}_1)k + \mathbf{g}v_1)v_2 \\ &= (p \underbrace{(\mathbf{q}_2\mathbf{A} + \mathbf{q}_1v_2)}_{\mathbf{q}_{mult}} + \underbrace{(\mathbf{r}_2\mathbf{A} + \mathbf{r}_1v_2)}_{\mathbf{r}_{mult}})k + \mathbf{g}v_1v_2 \end{aligned}$$

Therefore, the homomorphic multiplication yields a valid encryption of the product of the messages.

## A.2 Noise growth of homomorphic operations

In this section we show that the noise in the ciphertexts grows basically additively when we perform any homomorphic operation, including products. Using the analysis done in Section 4.3, it is easy to derive upper bounds to the noise accumulated by the homomorphic operations.

**Lemma 10 (Noise of homomorphic additions).** *Let  $n$  be an integer bigger than or equal to 2. For  $i \in \llbracket 1, n \rrbracket$ , let  $c_i$  be a scalar encryption of  $s_i$  and  $\mathbf{c}_i$  be a vector encryption of  $v_i$ . Compute the homomorphic sum of these ciphertexts as follows:  $c := \sum_{i=1}^n c_i \in R$  and  $\mathbf{c} := \sum_{i=1}^n \mathbf{c}_i \in R^\ell$ . Then,  $\text{err}(c) = \sum_{i=1}^n \text{err}(c_i)$  and  $\text{err}(\mathbf{c}) = \sum_{i=1}^n \text{err}(\mathbf{c}_i)$ . In particular, if all  $c_i$ 's and  $\mathbf{c}_i$ 's are fresh ciphertexts, we have*

$$\|\text{err}(c)\| < n2^p \text{ and } \|\text{err}(\mathbf{c})\| < n2^p.$$

*Proof.* Because each  $c_i$  is of the form  $(pq_i + r_i + \lfloor p/t \rfloor s_i)k$ , it is clear that  $\text{err}(c) = \sum_{i=1}^n r_i = \sum_{i=1}^n \text{err}(c_i)$ . By Lemma 2, if all  $c_i$ 's are fresh ciphertexts, we have  $\|\text{err}(c)\| \leq \sum_{i=1}^n \|\text{err}(c_i)\| < n2^p$  and the particular case holds.

Basically the same argument holds for vector ciphertexts.  $\square$

The noise growth of a sequence of homomorphic products involving only vector ciphertexts is essentially equal to the one of mixed products.

**Lemma 11 (Noise growth of products of vector ciphertexts).** *Let  $n$  be an integer bigger than or equal to 1. For  $i \in \llbracket 0, n \rrbracket$ , let  $\mathbf{c}_i$  be an encryption of  $m_i$ . Let also  $\mathbf{c}'_0 := \mathbf{c}_0$  and  $\mathbf{c}'_i := \text{GAHE.MultVec}(\mathbf{c}'_{i-1}, \mathbf{c}_i)$  for  $i > 0$ . (Notice that  $\mathbf{c}'_i$  is an encryption of  $\prod_{j=0}^i m_j$ ). Assume that  $B$  is an upper bound to the product of the plaintexts, i.e.,  $\left\| \prod_{i=j}^n m_i \right\| \leq B$  for  $0 \leq j \leq n$ . Then,*

$$\|\text{err}(\mathbf{c}'_n)\| < NB \|\text{err}(\mathbf{c}_0)\| + \sum_{i=1}^n N^2 B \ell b \|\text{err}(\mathbf{c}_i)\|.$$

*In particular, if all the products only involve fresh ciphertexts, then*

$$\|\text{err}(\mathbf{c}'_n)\| < 2N^2 B \ell b n 2^p.$$

*Proof.* This proof is basically equal to the one of Lemma 5, hence, we omit it.  $\square$

## B Applying multi-value bootstrapping to DGHV like schemes

In this section, we discuss briefly how our multi-value bootstrapping procedure can be applied to schemes like [DGHV10], which multiply the noise by  $t$  instead of shifting the message by some  $\alpha$ . Hence, consider that  $m \in \mathbb{Z}_t$  is encrypted as  $c = (\bar{p}q + tr + m)k \in \bar{R}$ . For the original [DGHV10], just consider  $\bar{N} = \bar{k} = 1$ .

We can transform  $c$  into a ciphertext  $\bar{c} = (\bar{p}\bar{q} + \bar{r} + \bar{\alpha}m)\bar{k}$ , which has the format expected by our refreshing procedure, as follows:

Let  $e := -\bar{p} \bmod 2t$  and  $\beta := \frac{(\bar{p}+e)}{t} \in \mathbb{Z}$ . Notice then, that

$$g^{-1}(c)\mathbf{g} \cdot \beta = c\beta = (\bar{p}\beta q + t\beta r + \beta m)\bar{k} = \left( \underbrace{\bar{p}(\beta q + r)}_{\bar{q}} + \underbrace{er + em}_{\bar{r}} + \underbrace{(p/t)m}_{\bar{\alpha}m} \right) \bar{k}.$$

Therefore, it is sufficient to append the following “transformation key” to the bootstrapping keys:

$$\mathbf{tk} := (\bar{p}\mathbf{q} + \mathbf{r})\bar{k} + \mathbf{g} \cdot \beta \in \bar{R}^{\ell}.$$

Besides that, we would need define  $\mathfrak{K}_i$  with  $\alpha := 1$  and all noise terms generated by `GAHE.EncScalar` and `GAHE.EncVec`, and also the noise terms of  $\mathbf{ek}_i$  would be multiplied by  $t$ , so that the extracted messages have the correct format at the end of the refreshing procedure.