Fully homomorphic encryption modulo Fermat numbers

Antoine Joux

Chaire de Cryptologie de la Fondation SU
Sorbonne Université
Institut de Mathématiques de Jussieu–Paris Rive Gauche
CNRS, Univ Paris Diderot.
Campus Pierre et Marie Curie, Paris

Abstract. In this paper, we recast state-of-the-art constructions for fully homomorphic encryption in the simple language of arithmetic modulo large Fermat numbers. The techniques used to construct our scheme are quite standard in the realm of (R)LWE based cryptosystems. However, the use of arithmetic in such a simple ring greatly simplifies exposition of the scheme and makes its implementation much easier.
In terms of performance, our test implementation of the proposed scheme is slower than the current speed records but remains within a comparable range. We hope that the detailed study of our simplified scheme by the community can make it competitive and provide new insights into FHE constructions at large.

1 Introduction

Fully homomorphic encryption (FHE) is an extremely fascinating concept. It aims at allowing any party to perform arbitrary computations on encrypted data without giving them the ability to decrypt. The concept was initially proposed by Rivest, Adleman and Dertouzos [RAD78] in 1978. For a long time it seemed to be an inaccessible grail. However, in 2009, Gentry proposed the first viable approach to FHE [Gen09] and invented the notion of bootstrapping which allowed him to offer (at least in principle) the ability to perform computations of unlimited depth.

In addition, Gentry explains in [Gen09] how to express the security of FHE schemes. In fact, he describes how it can be reduced to the traditional notion of semantic security. However, semantic security is often considered as insufficient for regular cryptosystems and it is usually required to preserve security in the presence of chosen ciphertext attacks (CCA). Unfortunately, it well-known that, essentially by definition, FHE schemes cannot resist the strongest form of the attacks, i.e., CCA2 attacks. Furthermore, when circular security is involved, even CCA1 security is problematic. This is a complex issue and recent developments (see [EHN+18]) provide innovative solutions to the problem.

Gentry's original system is based on ideal lattices. In an attempt to simplify the description of FHE, [vDGHV10] proposed another system based on the hardness of approximate GCD computations that works by performing arithmetic over large natural numbers. However, this conceptual simplicity goes with a high computational cost. After the initial breakthrough of Gentry, many other FHE systems were proposed, each offering new improvements in terms of key-sizes and efficiency. Note that most of these systems are based on the hardness of Learning-with-error (LWE) or Ring-LWE. Unfortunately, the gain in efficiency comes with the cost of using more complex mathematical objects which can make the underlying techniques harder to understand for a non-specialist audience.

In this paper, we propose to recast state-of-the-art FHE cryptosystems in the simple language of arithmetic modulo a large Fermat number. In this language, everything can be expressed in terms of numbers and their binary decompositions which are familiar to a broad audience. As a consequence, we achieve a simplicity of description close to FHE over the integers, and the system is easy to implement by just using a large number library. To maintain the simplicity, we only aim at semantic security for our proposal. In addition, despite the fact that our test implementation is slower than the fastest available system of [CGGI18], it has reasonably good performance.

1.1 Organization of the Paper

The paper is organized as follows. Section 2 describes the cryptosystem and its founding principles. Section 3 studies its correctness. In Section 4, we discuss cryptanalytic approaches and provide (hopefully) secure parameter choices.

Section 5 presents a basic experiment with a test implementation of the system.

2 Description of the cryptosystem

We organize the description of our FHE cryptosystem in three subsections. The first presents the system parameters and the general logic behind Fermat FHE. The second describes the functionalities which involve the private key of the system and are reserved to the key's owner. Finally, the third subsection describes the functionalities that involve encrypted data only and are intended for remote use.

2.1 Parameters and principle of the system

The first parameter is the Fermat number which serves as the main modulus for all computations:

$$F = 2^{2^f} + 1.$$

Since all numbers modulo F but -1 can be represented by 2^f-bit integers, we can choose to view them as juxtapositions of $L = 2^\ell$ blocks of $H = 2^h$ bits, where $\ell + h = f$. We remark $2^{LH} \equiv -1 \pmod{F}$.

As many FHE systems, we deal with noisy messages. In our case, the high bits of each block are used to hold significant bits, while the low bits contain noise. A fundamental identity that makes the system work is that given two bits x and y, we have:

$$x + y = 2(x \wedge y) + (x \oplus y).$$

Thus, if we can add the values of two bits as integers, or even as integers modulo 4, we are simultaneously computing an `AND` and a `XOR` gate. As a consequence, any cryptosystem that offers the possibility to homorphically add integers modulo 4, has the potential to become fully homomorphic since `AND` and `XOR` form a complete set of logical gates. However, after addition, the two resulting bits are stuck together and we cannot directly apply other consecutive gates. In the context of FHE, the idea that operations modulo 4 can be used to provide logical gates appeared in [DM15].

In order to correct the issue of having the `AND` and `XOR` of x and y stuck together, our system should provide the ability to extract them as separate bits. At the same time, the extraction procedure has to maintain the noise level of ciphertexts low enough to guarantee that consecutive gates operate properly. This is essentially the same idea as the gate bootstrapping of [CGGI18]. However, since bootstrapping as introduced in [Gen09] can be much more complex operation, we prefer for clarity's sake to call it *bit-extraction*.

2.2 Owner's functionalities (with secret key)

Private key generation. A private key of the system s is almost a randomly selected bitstring of L bits. More precisely, we insist that at most half the bits of s are set. This is easily done by generating s uniformly at random and by replacing s by its bitwise complement if it doesn't already satisfy the condition. This only reduces the effective keysize by one bit while permitting a better control of the noise during some operations on ciphertexts.

The individual bits are denoted s_i. Moreover, we associate to the key s an integer $\mathcal{S}$ modulo F defined as:

$$\mathcal{S} = \sum_{i=0}^{L-1} s_i 2^{iH}.$$

By abuse of notation, we often refer to $\mathcal{S}$ as the private key of the system and forget about the string s except when referring to the individual bits s_i. We denote the number of bits set to 1 in s as $\mathcal{N}_s$, i.e., we let $\mathcal{N}_s = \sum_{i=0}^{L-1} s_i$. Due to the constraint we have enforced when selecting s, we know that:

$$\mathcal{N}_s \leq L/2.$$

Encryption of zero. A noiseless encryption of zero would simply be a pair:

$$(A, A\mathcal{S} \pmod F)$$

computed from a random integer A modulo F. Of course, from such a noiseless encryption, it is easy to recover $\mathcal{S}$ – except for the trivial encryption $(0, 0)$. To prevent the recovery of $\mathcal{S}$, we add noise $\mathcal{E}$ to the second component, resulting in

$$(A, (A\mathcal{S} + \mathcal{E}) \bmod F).$$

The noise is created in two steps. First, we create a vector e of L coordinates which are taken independently from a probability distribution that we call the noise distribution $\hat{e}$. Second, we compute:

$$\mathcal{E} = \sum_{i=0}^{L-1} e_i 2^{iH}.$$

Knowing the private key, it is easy to create encryptions of zero and to test whether a pair (A, B) is an encryption of zero. This is done by computing $B - A\mathcal{S} \pmod F$ and by verifying that it can be decomposed as a sum of the form:

$$\sum_{i=0}^{L-1} e_i 2^{iH},$$

where the (signed) values e_i are small enough to be compatible with $\hat{e}$.

Note that, by linearity, a sum of two encryptions of zero is also an encryption of zero — with a slightly larger noise.

Concerning the noise distribution $\hat{e}$, we follow classical choices and two main options are possible. We take either a uniform distribution on a small interval around 0 or a discrete Gaussian distribution centered at zero and of small variance.

Integer encryption. Given an integer M modulo F, we encrypt it by taking a random encryption of zero (A, B) and by outputting $(A, B + M)$.

Integer (approximate) decryption. To decrypt a pair (A, B), we simply compute $\tilde{M} = B - A\mathcal{S} \pmod F$.

Note that the decryption $\tilde{M}$ of an encryption of M is equal to $M + \mathcal{E}$ where $\mathcal{E}$ is the error used while encrypting M. As a consequence, $\tilde{M}$ is an approximate or noisy decryption of M. In order to deal with exact values, we need to restrict the space of messages and to use an encoding scheme that permits to remove the noise after decryption. In the full system, we make use of two main coding schemes which are described in the following paragraphs.

Encoding of bits or bit-vectors Messages. First, we choose a high-order bit position to encode bit messages. Note that since we need at least one bit of margin to compute `AND` gates, we cannot encode on the highest bit of each block. When encoding input to basic gates, the simplest choice is to use bit position $b_M = H-2$. For inputs used in a richer way, such as Mux control bits or bit-extraction keys, the bits are encrypted with different (and possibly multiple) values of b_M.

The choice of b_M being given, the encryption of a bit $x \in \{0,1\}$ is simply an encryption of the integer $M_x = 2^{b_M} x$. This puts the bit at position b_M in the lowest block of the encrypted message. To perform decryption, one computes the approximate decryption $\tilde{M}_x$ and takes the closest integer to $\tilde{M}_x/2^{b_M}$. As long as the noise remains smaller than 2^{b_M-1}, the decrypted value is correct.

It is also easy to encrypt a vector of L bits in a single message. For this purpose, we just define the message as:

$$M_x = \sum_{i=0}^{L-1} x_i 2^{iH+b_M}.$$

We do not directly use this representation to encode bit vectors in the basic scheme, even if it could potentially be useful for SIMD encodings. However, this vector encoding is used during the bit-extraction procedure.

2.3 Remote functionalities (without secret key)

Integer addition. Given two encrypted integers, anyone can add then and obtain an (approximate) encryption of their sum with a slightly larger noise than the input encryptions. Given the encryption of two encoded bits (using the same value for b_M), this, in particular, computes their sum modulo 4. As a consequence, the resulting encrypted value contains both the `XOR` and `AND` of the input bits. These bits can then be separated by the bit-extraction procedure that appears below.

Mux gates. In order to implement the bit-extraction procedure, we need a subroutine that given an encrypted bit c and the encryption of two integers I_0 and I_1 produces another encryption of I_c, possibly with larger noise. Let the encryption of I_0 (resp. I_1) be given as a pair $(A^{(0)}, B^{(0)})$ (resp. $(A^{(1)}, B^{(1)})$). Remark that $(A^{(2)}, B^{(2)}) = (A^{(1)} - A^{(0)}, B^{(1)} - B^{(0)})$ is an encryption of $I_1 - I_0$. We first compute $(A^{(3)}, B^{(3)})$ an encryption of $c(I_1 - I_0)$ and then output $(A^{(0)} + A^{(3)}, B^{(0)} + B^{(3)})$ as the final encryption of $I_c = I_0 + c(I_1 - I_0)$.

The computation of $A^{(3)}$ and $B^{(3)}$ is performed from a special encoding of the bit c that we describe below. In order to do the computation, we first decompose $A^{(2)}$ and $B^{(2)}$ in binary, denoting each individual bit as $a_i^{(2)}$ and $b_i^{(2)}$. We then regroup the bits according to their position inside of their block, computing:

$$A_i^{(2)} = \sum_{j=0}^{L-1} a_{jH+i}^{(2)} 2^{jH},$$

and similarly for $B_i^{(2)}$. Of course, we have:

$$A^{(2)} = \sum_{i=0}^{H-1} A_i^{(2)}\, 2^i \quad \text{and} \quad B^{(2)} = \sum_{i=0}^{H-1} B_i^{(2)}\, 2^i.$$

In addition, each of the numbers $A_i^{(2)}$ and $A_i^{(2)}$ are small according to the way we express the size of noise. Indeed, in each block the value is either 0 or 1. Because of this, the noise on the resulting encryption can be controled. A detailed analysis of the noise is given in Section 3.

Assume now that we are given encryptions of $2^i c$ in the form (K_i, L_i) and encryptions of $-2^i c\mathcal{S}$ in the form (M_i, N_i) for all $i \in [0, H-1]$, then we can compute $(A^{(3)}, B^{(3)})$ as:

$$\begin{aligned} A^{(3)} &= \sum_{i=0}^{H-1} B_i^{(2)}\, K_i + \sum_{i=0}^{H-1} A_i^{(2)}\, M_i \quad \text{and} \\ B^{(3)} &= \sum_{i=0}^{H-1} B_i^{(2)}\, L_i + \sum_{i=0}^{H-1} A_i^{(2)}\, N_i. \end{aligned}$$

Indeed, if c is zero, this produces a sum of product of encryption of zeroes by values which are small. Thus, in that case, we have an encryption of 0. When c is one, the computation has two parts, a linear combination of multiples of encryption of 0 as before and a message which is the linear combination with the same coefficients of the message values of the combined ciphertext, i.e., the message part is equal to:

$$\begin{aligned} &\sum_{i=0}^{H-1} B_i^{(2)}\, 2^i c - \sum_{i=0}^{H-1} A_i^{(2)}\; 2^i c\mathcal{S} \\ &= c(B^{(2)} - A^{(2)}\mathcal{S}). \end{aligned}$$

This is precisely a reencryption of (c times) the (approximate) decryption of $(A^{(2)}, B^{(2)})$. Thus, this computation on the binary decomposition achieves the desired functionnality. For completeness, the corresponding pseudo-code is given as Algorithm 1.

Note: The creation of all the encryptions of the value $2^i c$ and $2^i c\mathcal{S}$ calls for two important comments:

1. Since we know that $(2A, 2B)$ is an encryption of the double of the value encrypted in (A, B), one might wonder why we create all the encryptions of $2^i c$ rather than just an encryption of c. This comes from the need to control the resulting noise value. Indeed, every doubling of the encrypted value also doubles the size of the noise. In truth, depending on the exact parameters of the system and the analysis of the noise, only a fraction of these encryptions of $2^i c$ might be necessary. Also note that for small values of i, it is even

possible to remove the corresponding components from the computation of $(A^{(3)}, B^{(3)})$, since these contributions are submersed in the noise.

In summary, it is possible to reduce the size of an encrypted mux, at the cost of degrading the noise propagation property. This leads to different compromises for the system parameters.

2. It is interesting to remark that an encryption of $-2^i c\mathcal{S}$ can be built in two distinct ways. One option is to add $-2^i c\mathcal{S}$ to the B-part of an encryption of zero as for any encryption. The other is to add $2^i c$ to the A-part of an encryption of zero, resulting in the same distribution of outputs.

   This could be important when the FHE system is used by more than two parties. Indeed, it allows users without knowledge of the key $\mathcal{S}$ to encrypt a mux-key of their choice. However, we need to provide them with a method to construct encryptions with a small enough noise level to be used as Mux keys. This would require specific adjustments to the system parameters.

Algorithm 1: Encrypted Mux Algorithm.

Input: Ciphertexts $(A^{(0)}, B^{(0)})$, $(A^{(1)}, B^{(1)})$
Encrypted Mux c as (K_i, L_i) and (M_i, N_i) for i in $[0 \cdots H-1]$
Result: Re-encryption of $(A^{(c)}, B^{(c)})$

$(A^{(2)}, B^{(2)}) \leftarrow (A^{(1)} - A^{(0)} \pmod F, B^{(1)} - B^{(0)} \pmod F)$;
if $A^{(2)} = F-1$ **then** $A^{(2)} \leftarrow 0$;
if $B^{(2)} = F-1$ **then** $B^{(2)} \leftarrow 0$;
Binary decompose $A^{(2)}$ as $\sum_{i=0}^{LH-1} a_i\, 2^i$;
Binary decompose $B^{(2)}$ as $\sum_{i=0}^{LH-1} b_i\, 2^i$;
$(A^{(3)}, B^{(3)}) \leftarrow (0, 0)$;
for $i = 0$ **to** $H-1$ **do**
  $\mathcal{A} \leftarrow \sum_{j=0}^{L-1} a_{jH+i}\, 2^{jH+i}$;
  $\mathcal{B} \leftarrow \sum_{j=0}^{L-1} b_{jH+i}\, 2^{jH+i}$;
  $A^{(3)} \leftarrow A^{(3)} + \mathcal{B}\, K_i + \mathcal{A}\, M_i$;
  $B^{(3)} \leftarrow B^{(3)} + \mathcal{B}\, L_i + \mathcal{A}\, N_i$;
end
return $(A^{(3)} + A^{(0)}, B^{(3)} + B^{(0)})$

Bit-extraction procedure. As mentioned in Section 2.1, one of the main functionality is the extraction of a bit from an encrypted message into a fresh encryption. At the same time, this should limit the noise level as the new ciphertext. As a result, the newly encrypted version of the bit can be used in further computations. Interestingly, this bit-extraction procedure can also change the encryption and even the parameters of the system.

In the simplest setting, we directly use a circular encryption paradigm and both sets are keys and parameters are identical. However, it is possible to use the ability to change keys and parameters in variations of the system.

To make this explicit, thoughout the section, we denote by F_{in}, L_{in}, H_{in} and $\mathcal{S}_{\text{in}}$ the parameters and key of the encrypted message (A, B) given as input. Similarly, F_{out}, L_{out}, H_{out} and $\mathcal{S}_{\text{out}}$ are the parameters and key of the message produced by the bit-extraction.

More precisely, we assume that we want to extract from (A, B) the bit x at position b_p in the lowest order block of the decryption. As a consequence, we have:

$$B - \mathcal{S}_{\text{in}} A = Y\, 2^{b_p+1} + x\, 2^{b_p} + e_{\text{in}} \pmod{F_{\text{in}}}$$

with $x \in \{0, 1\}$. The value $Y < F_{\text{in}} \cdot 2^{-b_p}$ can be arbitrary (since we are ignoring the values in other blocks) and e_{in} denotes the error on the lowest block.

The goal is to produce a new encryption $(A_{\text{out}}, B_{\text{out}})$ such that:

$$B_{\text{out}} - \mathcal{S}_{\text{out}} A_{\text{out}} = Y'\, 2^{H_{\text{out}}} + x\, 2^{b_M} + e_{\text{out}},$$

under the output key. The value Y' can be an arbitrary value $Y' < F_{\text{out}} \cdot 2^{-H_{\text{out}}}$. Indeed, we only focus on the content of the lowest block of output. In order to be able to re-use the new message in further operations, we need to bound the error e_{out}. Also note that the position b_M doesn't need to be the same as the position b_P and that the form of new encryption ensures that there are only zeroes in the lowest order block above the encrypted bit in position b_M.

As in [DM15], we rely on the use of an homomorphic accumulator in order to simulate an approximation of the exact decryption process. More precisely, as in [CGGI18], we perform this computation modulo 2^{b_p+1}. The modulus is chosen to remove the parasitical high bits contained in Y. Recall that the exact decryption process would compute $V = B - \mathcal{S}_{\text{in}} A \pmod F$, then $V_0 = V \pmod{2^{H_{\text{in}}}}$ and finally round $V_0/2^{b_p}$ to the closest integer.

Decomposing A and B into their blocks as:

$$A = \sum_{i=0}^{L_{\text{in}}-1} A_i 2^{iH_{\text{in}}} \quad \text{and} \quad B = \sum_{i=0}^{L_{\text{in}}-1} B_i 2^{iH_{in}},$$

we can rewrite this as:

$$V_0 = B_0 - A_0 s_0 + \sum_{i=1}^{L_{\text{in}}-1} A_i s_{L_{\text{in}}-i} + C \pmod{2^{H_{\text{in}}}},$$

where the value s_i are the bits of $\mathcal{S}_{\text{in}}$ and C is a carry value induced by reduction modulo the Fermat number. Since the key $\mathcal{S}_{\text{in}}$ comes from a binary string, we can bound C and write $|C| \leq \mathcal{N}_s + 1$. More precisely, here $\mathcal{N}_s$ denotes the number of 1s in $\mathcal{S}_{\text{in}}$ and we have $\mathcal{N}_s \leq L_{\text{in}}/2$. In particular, this ensures that $|C| \leq L_{\text{in}}$. Note that the sign change between A_0 and the others A_i also arises from the reduction modulo F_{in}.

By design, V_0 is equal to $Y \cdot 2^{b_p+1} + x \cdot 2^{b_p} + E_0$, where x is the bit we want to decrypt, b_p the position where the bit is located and E_0 is an additional term that regroups e_{in}, C and possibly message bits at a position below b_P.

To decrypt the value of a `XOR` gate, it thus suffices to compute:

$$W_0 = B_0 - A_0 s_0 + \sum_{i=1}^{L_{\text{in}}-1} A_i s_{L_{\text{in}}-i} \pmod{2^{b_P+1}},$$

which is an approximation of $V_0 \pmod{2^{b_P+1}}$, within C. If we assume that $|E_0|+|C|$ it smaller than 2^{b_P-1}, then the bit x is the closest integer to $W_0/2^{b_P}$.

However, when decrypting the result of an `AND` gate, there is a bit at the position immediately below corresponding to the `XOR` value. As a consequence, $W_0/2^{b_P}$ can close to 0, $1/2$ or 1. The first two cases correspond to a 0 and the last to a 1. Unfortunately, directly rounding $1/2$ to the closest integer would lead to an undefined result. To avoid this issue, we have two options. The simplest (and slowest) approach is to start by extracting the `XOR` value, in the same bit position as the original message bit. Subtracting this encryption of the extracted `XOR` from the base encrypted message removes this bit. Once this is done, the `AND` value becomes isolated and can be normally extracted. The drawback of this approach is that it requires two consecutive bit-extractions, which slows the computation down by a factor of two.

The second approach to extract an `AND` bit, is to perform a single extraction where W_0 is replaced by $W_0' = W_0 - 2^{b_P-2}$ before taking the highest bit. After the subtraction, $W_0'/2^{b_P}$ is close to $-1/4$, $1/4$ or $3/4$. Thus, the closest integer corresponds to the expected bit value in the three cases. The drawback here is that the noise analysis becomes more difficult.

The key to performing modular computations with encrypted values comes from [AP14], which embeds modular integers into a symmetric group. The idea is simplified in [DM15], which makes use of roots of unity inside the ring of polynomials they are using.

Basically, the idea is to use a rotating buffer containing 2^{b_P+1} binary values. Initially, the vector contains a value z_0 at all positions closer to position 0 than position 2^{b_P} and a value z_1 in the other positions. This encrypted vector is then rotated by an amount equal to W_0 (or $W_0' = W_0 - 2^{b_P-2}$). The effect is to bring one copy of the desired value z_x in position 0 of the vector.

Implementing this idea within the FHE scheme requires some tricks. First, since $\mathcal{S}_{\text{in}}$ needs to be kept secret W_0 cannot be computed explicitly. Thus, instead of rotating the buffer all at once, we proceed in an incremental way. For each rotation by an offset $A_i s_{L_{\text{in}}-i}$, we rotate by A_i and use a multiplexer operation keyed by an encrypted $s_{L_{\text{in}}-i}$ to keep the rotated copy when $s_{L_{\text{in}}-i} = 1$ and the original otherwise.

The second trick allows to greatly reduce the cost by packing the rotating buffer into a vector inside a single encrypted number. This fixes the size of the

rotating buffer to $2L_{\text{out}}$. Indeed, a one-block rotation corresponds to multiplication by $2^{H_{\text{out}}}$ and performing $2L_{\text{out}}$ such multiplications brings us back to the original value modulo F_{out}. In other words, $2^{H_{\text{out}}}$ is a $2L_{\text{out}}$-th root of unity, which allows us to proceeds as in [DM15].

In order to perform the computations modulo $2L_{\text{out}}$ instead of 2^{b_P+1}, we need to rescale all the numbers in the computation of W_0 by a factor $2^{\ell_{\text{out}}-b_P}$ and round to the closest integer. This introduces some additional noise but the principle remains the same and works as long as we control the size of the noise. More precisely, assuming that $\ell_{\text{out}} \leq b_P$ we are thus computing:

$$\mathcal{W}_0 = \lfloor \frac{B_0 - s_0 A_0}{2^{b_P-\ell_{\text{out}}}} \rceil + \sum_{i=1}^{L_{\text{in}}-1} \lfloor \frac{A_i}{2^{b_P-\ell_{\text{out}}}} \rceil s_{L_{\text{in}}-i} + (-2^{\ell_{\text{out}}-2}) \pmod{2L_{\text{out}}},$$

where the parenthesized term $-2^{\ell_{\text{out}}-1}$ is only included when computing an `AND` gate with the second method. The grouping of $B_0 - s_0 A_0$ inside a single rounding might seem surprising at first; it can be easily achieved by applying a Mux controled by s_0 to select between a rotation by $\lfloor B_0\, 2^{\ell_{\text{out}}-b_P} \rceil$ and a rotation by $\lfloor (B_0 - A_0)\, 2^{\ell_{\text{out}}-b_P} \rceil$

In addition, packing the rotating buffer into a single vector forces us to choose $z_1 = -z_0$. Indeed, we can only store values in the first L_{out} blocks, the other being implicitly defined by rotation by L_{out} blocks which multiplies by -1 modulo the Fermat number.

If we choose $z_0 = -2^{b_M-1}$, the produced ciphertext contains a value equal to $(-1)^{x+1}\, 2^{b_M-1}$ in its lowest order block. Adding 2^{b_M-1} to the B-part of the ciphertext transforms this value into $x\, 2^{b_M}$ as desired. In the pseudo-code given in Algorithm 2, the initial of z_0 and z_1 in their respective blocks is done by setting all blocks to 2^{b_M-1} and applying a well-chosen (fixed) rotation in order to have z_0 around block 0.

Basic (circular) gates. With the bit-extraction at hand, applying the `XOR` and `AND` gates is easy. We assume here that we are using the circular setting, with the same key, parameters and message bit position for all encryptions.

Given encryptions of two bits x and y, placed at position b_M, we simply compute their sum and apply bit-extraction to obtain both the `XOR` and the `AND` of x and y. The first method is to first extract an encrypted copy of the `XOR` (using $b_P = b_M$), subtract it from the original value and extract the `AND` value (using $b_P = b_M + 1$).

The second method is to extract the `AND`, using $b_P = b_M + 1$ and the subtraction of 2^{b_P-2} from W_0 as explained above. Once an encrypted copy of the `AND` is extracted, it suffices to subtract its double from the original sum to obtain an encryption of the `XOR` bit. This leads to a higher noise value in the `XOR` but computes both gates with a single extraction. If this optimization is used, it is necessary to track the noise growth and apply bit-extraction on the `XOR` thus obtained when the noise becomes critically large.

Algorithm 2: Bit-Extraction Algorithm.

Input: Ciphertext (A, B), bit position to extract b_P, Boolean `IsIsolated`, new bit position b_M
Encrypted Mux Description of all key bits s_i for i $[0 \cdots L_{\text{in}} - 1]$.
Result: Re-encryption of bit from position b_P at position b_M as $(A_{\text{out}}, B_{\text{out}})$

Block decompose A as $\sum_{i=0}^{L_{\text{in}}-1} A_i \, 2^{iH_{\text{in}}}$;
Block decompose B as $\sum_{i=0}^{L_{\text{in}}-1} B_i \, 2^{iH_{\text{in}}}$;
$(A', B') \leftarrow (0, \sum_{i=0}^{L_{\text{out}}-1} 2^{iH_{\text{out}}+b_M-1})$;
if `IsIsolated` **then** $v_{\text{offset}} \leftarrow 2^{\ell_{\text{out}}-1}$ **else** $v_{\text{offset}} \leftarrow 2^{\ell_{\text{out}}-2}$;
$v_0 \leftarrow \lfloor \frac{B_0}{2^{b_P-\ell_{\text{out}}}} \rceil + v_{\text{offset}} \pmod{2L_{\text{out}}}$;
$v_1 \leftarrow \lfloor \frac{B_0-A_0}{2^{b_P-\ell_{\text{out}}}} \rceil + v_{\text{offset}} \pmod{2L_{\text{out}}}$;
$(A^{(0)}, B^{(0)}) \leftarrow \left(\left(2^{v_0 H_{\text{out}}} A'\right) \bmod F_{\text{out}}, \left(2^{v_0 H_{\text{out}}} B'\right) \bmod F_{\text{out}} \right)$;
$(A^{(1)}, B^{(1)}) \leftarrow \left(\left(2^{v_1 H_{\text{out}}} A'\right) \bmod F_{\text{out}}, \left(2^{v_1 H_{\text{out}}} B'\right) \bmod F_{\text{out}} \right)$;
$(A_{\text{out}}, B_{\text{out}}) \leftarrow \texttt{Mux}\left(s_0, (A^{(0)}, B^{(0)}), (A^{(1)}, B^{(1)})\right)$;
for $i = 1$ **to** $H_{in} - 1$ **do**
  $v_1 \leftarrow \lfloor \frac{A_i}{2^{b_P-\ell_{\text{out}}}} \rceil \pmod{2L_{\text{out}}}$;
  $(A^{(1)}, B^{(1)}) \leftarrow \left(\left(2^{v_1 H_{\text{out}}} A_{\text{out}}\right) \bmod F_{\text{out}}, \left(2^{v_1 H_{\text{out}}} B_{\text{out}}\right) \bmod F_{\text{out}} \right)$;
  $(A_{\text{out}}, B_{\text{out}}) \leftarrow \texttt{Mux}\left(s_{L_{\text{in}}-i}, (A_{\text{out}}, B_{\text{out}}), (A^{(1)}, B^{(1)})\right)$;
end
return $(A_{\text{out}}, (B_{\text{out}} + 2^{b_M-1}) \bmod F_{\text{out}})$

Also note that the `NOT` gate is very easy to compute. If (A, B) is an encryption of x at position b_M then $(A, 2^{b_M} - B \pmod F)$ is an encryption of $\bar{x} = 1 - x$. Moreover, this operation does not increase the noise level.

We thus have a complete set of gates and can perform universal computations with circuits of arbitrary depth. In addition to the optimization presented above, many others are possible. It is also possible to directly compute additional gates without needing to expand them in terms of the basic gates. We give examples in the Appendix.

3 Details and analysis of remote functionalities

In this section, we study in details the noise value that arise from the remote functionalities performed with the secret key. The analysis can be done with two scenarii in mind. We either request an absolute guarantee that the errors always stay small enough to permit correct decryption. With this first scenario, the natural choice of noise distribution is the uniform distribution in an interval around zero. This gives an upper-bound on the initial noise level and the goal of the analysis is to provide similar upper-bounds on the noise of subsequent ciphertexts.

Alternatively, we can allow a vanishingly small probability of error during computation in order to get smaller parameters for the system. With this scenario, the natural choice of noise distribution is a discrete Gaussian around 0 with given variance. Unfortunately, this second scenario requires assumptions about the distribution of errors resulting from all computations on ciphertext. Essentially, it needs to assume that every resulting noise follows a Gaussian (or sub-Gaussian as in [AP14]) distribution, with some bound on its variance and that the all the resulting noise values behave like independent variables. With this assumption, when noises are added, a bound on the variance of the sum is obtained by summing the bounds on the variances of the summands. The expectations can also be added to get the expectation of the sum (but these are usually all equal to zero). If the allowed error for correctness is many standard deviations away, we obtained a very small (and effectively computable) probability of getting an erroneous result. However, being heuristic, this approach presents some risks and also leads to a complex analysis in order to obtain concrete probabilities of erroneous computation. Unfortunately this approach is necessary to obtain optimized parameter choices offering a reasonable efficiency for the FHE scheme.

Also, note that, in both cases, noise analysis is done assuming that all computations are honesting performed. Indeed, it is extremely simple for a dishonest remote user to add arbitrary noise of any magnitude to the ciphertexts he computes. Furthermore, any information about these ciphertexts leaked by the secret owner could fully compromise the system. However, since, as discussed in the introduction, we only consider semantic security, we will not consider these attacks any further.

3.1 Analysis of Mux

In order to perform this analysis, it simpler to explicitly compute the decryption of the resulting ciphertext and to give the exact formula for the output error. In order to do this, let's assume that the inputs satisfy:

$$\begin{aligned} B^{(i)} - A^{(i)}\mathcal{S} &\equiv I_i + \mathcal{E}_i \pmod F \quad \text{for } i = 0, 1, \\ L_i - K_i\mathcal{S} &\equiv 2^i c + \mathcal{F}_i \pmod F \quad \text{for } i = 0 \cdots H-1, \text{ and} \\ N_i - M_i\mathcal{S} &\equiv -2^i c\mathcal{S} + \mathcal{G}_i \pmod F \quad \text{for } i = 0 \cdots H-1. \end{aligned}$$

$$\begin{aligned} A^{(3)} &= \sum_{i=0}^{H-1} B_i^{(2)} K_i + \sum_{i=0}^{H-1} A_i^{(2)} M_i \quad \text{and} \\ B^{(3)} &= \sum_{i=0}^{H-1} B_i^{(2)} L_i + \sum_{i=0}^{H-1} A_i^{(2)} N_i. \end{aligned}$$

With these notations, we have:

$$\begin{aligned} B^{(3)} - A^{(3)}\mathcal{S} &\equiv \sum_{i=0}^{H-1} B_i^{(2)}\,(L_i - K_i\mathcal{S}) + \sum_{i=0}^{H-1} A_i^{(2)}\,(N_i - M_i\mathcal{S}) \\ &\equiv \sum_{i=0}^{H-1} B_i^{(2)}\,(2^i c + \mathcal{F}_i) + \sum_{i=0}^{H-1} A_i^{(2)}\,(-2^i c\mathcal{S} + \mathcal{G}_i) \\ &\equiv c(B^{(2)} - A^{(2)}\mathcal{S}) + \sum_{i=0}^{H-1} B_i^{(2)}\,\mathcal{F}_i + \sum_{i=0}^{H-1} A_i^{(2)}\,\mathcal{G}_i \\ &\equiv c(I_1 - I_0 + \mathcal{E}_1 - \mathcal{E}_0) + \sum_{i=0}^{H-1} B_i^{(2)}\,\mathcal{F}_i + \sum_{i=0}^{H-1} A_i^{(2)}\,\mathcal{G}_i \pmod F \end{aligned}$$

After adding back $(A^{(0)}, B^{(0)})$, we obtain a message that decrypts exactly to:

$$I_c + \mathcal{E}_c + \sum_{i=0}^{H-1} B_i^{(2)}\,\mathcal{F}_i + \sum_{i=0}^{H-1} A_i^{(2)}\,\mathcal{G}_i.$$

Thus, if the maximum error values in each block of $\mathcal{E}_i$, $\mathcal{F}_i$ and $\mathcal{G}_i$ (for all relevant i) are respectively bounded by $E_{\max}$, $F_{\max}$ and $E_{\max}$, the error on blocks of the final message is bounded by:

$$E_{\max} + HL\,F_{\max} + HL\,G_{\max}.$$

With the (sub)Gaussian modelling of errors, we need to sum variances instead of bounds in the above formula.

3.2 Correctness of bit-extraction

As seen in the description of the bit-extraction procedure, exact decryption of the low-order block computes

$$V_0 = b_0 - a_0 s_0 + \sum_{i=1}^{L-1} a_i s_{L_{\text{in}}-i} + C \pmod{2^B},$$

with a carry C such that $|C| \leq \mathcal{N}_s + 1$.

By design, V_0 is equal to $x \cdot 2^{b_p} + E_0$, where x is the bit we want to decrypt, b_p the position where the bit is located and E_0 some error term.

During bit-extraction of an isolated bit, i.e. a `XOR` or an `AND` with the first method, we extract from the rotating buffer the value in position:

$$\mathcal{W}_0 = \lfloor \frac{B_0 - s_0 A_0}{2^{b_P - \ell_{\text{out}}}} \rceil + \sum_{i=1}^{L_{\text{in}}-1} \lfloor \frac{A_i}{2^{b_P - \ell_{\text{out}}}} \rceil s_{L_{\text{in}}-i} \pmod{2L_{\text{out}}},$$

The extracted value is correct, if and only if, $\lfloor \mathcal{W}_0 / L_{\text{out}} \rceil = x$. This is equivalent to checking that the total accumulated error is smaller than $2^{\ell_{\text{out}}-1}$.

Since every rounding induces an error of at most $1/2$ in absolute value, the total error is:

$$\mathcal{N}_s/2 + (|E_0| + |C|)/2^{b_P - \ell_{\text{out}}}.$$

Since we enforced the private key $\mathcal{S}$ to contain at most 50% of 1s during key generation, we have $\mathcal{N}_s/2 \leq 2^{\ell_{\text{in}}-2}$. Thus, if $\ell_{\text{out}} \geq \ell_{\text{in}}$, the extracted value is correct as long as:

$$|E_0| \leq 2^{b_P-2} - |C|2^{\ell_{\text{out}}-b_P}. \leq 2^{b_P-2} - 2^{\ell_{\text{in}}+\ell_{\text{out}}-b_P}.$$

In particular, if we choose the parameters to satisfy $\ell_{\text{in}} + \ell_{\text{out}} \leq 2b_P - 3$ then it suffices to have $|E_0| \leq 2^{b_P-3}$. In the standard circular case, since b_P is either b_M or $b_M + 1$ and $\ell_{\text{in}} = \ell_{\text{out}} = \ell$, it suffices to require $\ell \leq b_M - 2$.

However, this reasoning isn't sufficient when applying the second method to compute an `AND` gate. Indeed, the subtraction of $2^{\ell_{\text{out}}-2}$ introduces an extra error term which added to the above bound on the rounding error is enough to corrupt bit values. There are several ways out of the problem:

1. Use a larger value for ℓ_{out} than for ℓ_{in} to get additional margin. This can be costly because of the use of larger parameters. In addition, it requires a keyswitch procedure (see the Appendix) to convert back to the input parameters.
2. Make the key sparser, for example choosing $\mathcal{N}_s \leq L_{\text{in}}/4$. This is easy to implement but makes the security analysis harder in order to account for the possibility of dedicated attacks on sparse keys.
3. Provide a better bound on the rounding error. This is not possible directly since the bound is tight for specifically constructed ciphertexts. However, at runtime, the implementation knows the rounding error on each of the L_{in} terms of $\mathcal{W}_0$. Since the global error is a sum of at most $L_{\text{in}}/2$ of these terms, it suffices to compute the sum of the $L_{\text{in}}/2$ largest positive terms (or of all positive terms if there are fewer than that) and a similar sum for the largest negative terms (taken in absolute value). The largest of these values gives a concrete runtime upper bound.
   On average, we expect this runtime bound to be of the order of $L_{\text{in}}/8$ (instead of $L_{\text{in}}/4$ for the basic bound). This provides enough margin to permit bit-extraction with a somewhat stricter bound on $|E_0|$. For example, if the parameters satisfy $\ell_{\text{in}} + \ell_{\text{out}} \leq 2b_P - 4$ then the updated bound becomes $|E_0| \leq 2^{b_P-4}$. For an `AND` gate in the circular case, since b_p is $b_M + 1$, the condition $\ell \leq b_M - 2$ is left unchanged.
   Note that, in the rare event where bit-extraction is declared unsuccessful due to a rounding error bound being too large, it is possible to re-randomize the ciphertext by adding some encryption of zero and restart the procedure. Alternatively, one can also use the first method to recover from these rare error cases. Since bit-extraction fails very rarely, the additional computation cost is negligible when considering the global cost of the circuit computation.

4. We can go further in this direction by giving a probabilistic bound on the accumulated rounding error with the (sub)Gaussian model. Indeed, if we assume that the rounding errors are uniform and independent, the sum of $\mathcal{N}_s$ of them is a subgaussian distribution with a relatively small variance. Under these assumptions, we expect the concrete error magnitude to be even smaller than the worst-case bound for the isolated bit case. Using this approach makes the code simpler since there is no need to compute a concrete bound during the homomorphic computation. However, as we already mentionned, the analysis becomes messier and relies on unproven hypotheses about the distributions of errors. Yet, this approach is necessary to provide optimized parameters.

3.3 Noise on Extracted Result

The noise after the bit-extraction process comes from a succession of L_{in} Mux applied to a noiseless initial value. As a consequence, it is equal to the sum of noise introduced by each Mux.

The exact value of the bound on the sum greatly depends on which encrypted value of the form $2^i s_j$ and $-2^i s_j \mathcal{S}$ are provided for each key bit s_j. One extreme case is to assume that encryptions are provided for all powers 2^i with a noise bound ε_{key} then the total noise after bit-extraction is bounded by:

$$2L_{\text{in}} H_{\text{out}} L_{\text{out}} \varepsilon_{\text{key}}.$$

At the other extreme, we may only be given the encryptions of $2^{H_{\text{out}}/2} s_j$ and $-2^{H_{\text{out}}/2} s_j \mathcal{S}$, again with a noise bound ε_{key}. In that case, all the terms with $i < H_{\text{out}}/2$ in the Mux sums are removed, which induces an error equal to $2^i s_j$. Furthermore, terms with $i > H_{\text{out}}/2$ make use of scalings of the provided encryption and the noise is also scaled. Thus, the total error is bounded by:

$$2L_{\text{in}} 2^{H_{\text{out}}/2} \left(1 + L_{\text{out}} \varepsilon_{\text{key}}\right).$$

4 Security Evaluation and parameters

In section, we give a short analysis of known attacks that work on related systems and explain how to choose parameters to rule out basic adaptations of these attacks to our proposal. Of course, further cryptanalytic scrutiny by independent teams is necessary to give confidence in the system and quantify its security more precisely.

4.1 The underlying hard problems

Before going into the analysis of the attacks, we first state the hard problem(s) that underly Fermat FHE. More precisely, we focus on the distinguishing problems. In fact, we can state two distinct problems depending on whether we give out the full value of the B-part or just its low-order block.

Full encryptions of zero. The main problem on which the system relies is the hardness of distinguishing between encryptions of zero (with some given noise distribution) and purely random pairs of numbers. Equivalently, the adversary is given access to one of the two following black-boxes and needs to distinguish them. The first box simply generates pairs of uniform independent random number (A, B) modulo F. The second box contains a binary key s and generate pairs $(A, A\mathcal{S} + \mathcal{E})$ where $\mathcal{E}$ is picked from a fixed noise distribution known to the adversary.

Truncated encryptions of zero. Another problem related to the system is the same distinguishing problem where the second component of each pair is reduced to its low-order block. I.e., instead of providing B each box releases $B \pmod{2^H}$ for some fixed small H.

Any adversary with non-negligible advantage against this second problem can be adapted easily to solve the first one. Indeed, it suffices to truncate the provided samples to their low-order block on the B-part and apply the given distinguisher.

However, the converse is unlikely to hold. Indeed, the second problem presents similarities to LWE while the first is similar to RLWE.

4.2 Ruling out combinatorial attacks

Given an encrypted message, finding the corresponding noise value is enough to recover the secret key. Of course, directly guessing a secret candidate also suffices. As usual, a birthday algorithm performs better than exhaustive search and allows key recovery. If quantum algorithms are taken into account, a third-root time algorithm is expected.

To estimate the security level, we assume that the best combinatorial attack works in third-root of the minimum of the key space and noise space.

As a consequence, since the main parameters are powers of 2, for a 128-bit security level, we need at least $L = 512$ blocks; for a 256-bit security level, the minimum becomes $L = 1024$ blocks. With the third-root attack rule-of-thumb, this is even overkill.

In any case, these choices seem sufficient to rule-out any possiblity of a direct combinatorial attack.

4.3 Lattice-based attacks

There are lots of possible lattice-based attacks that have been proposed on (R)LWE systems. For our system, one can directly from an encryption (A, B) of

zero express the recovering of the secret key s and error e as a lattice problem by looking for the short solutions of the equation:

$$B - \sum i = 0^{L-1} s_i 2^{iB} A \equiv \sum i = 0^{L-1} e_i 2^{iB} \pmod{F}.$$

Another possibility given many encryption (A_i, B_i) of zero is to build a distinguisher by first finding a short solution of:

$$\sum x_i A_i \equiv 0 \pmod{F}$$

and then checking whether $\sum x_i B_i \pmod{F}$ contains small values in its basis 2^H-decomposition.

In both cases, we consider a lattice of determinant F and want to obtain vectors of norm shorter than 2^H. We now give a general analysis of the cost of lattice attack sharing these parameters.

Let L be a general lattice of determinant F and dimension d. We want to estimate the cost of finding a short vector of norm at most 2^H. There are several models to perform such estimates, a survey in the context of LWE is given in [APS15]. These models predict, for an arbitrary $\delta > 1$, the runtime to find a vector of norm $F^{1/d}\delta^d$.

The simplest is the Lindner-Peikert model [LP11] which predicts that the logarithm (in basis 2) of the running time (expressed in seconds on a 2.3 GHz computer) is:

$$\frac{1.8}{\log_2 \delta} - 110.$$

This simple model is described as *too optimistic* in [APS15]. The paper also give a slightly more complex quadratic model which approximates the logarithm of the running time by:

$$\frac{0.009}{\log^2 \delta} - 27,$$

For a fixed δ, the quantity $F^{1/d}\delta^d$ is minimized at $d = \sqrt{\log F / \log \delta}$. In order to have the value at this point to be smaller than 2^H, we need:

$$2\sqrt{\log_2 F \log_2 \delta} \leq H.$$

Thus, we require:

$$\log_2 \delta \leq \frac{H^2}{4 \log_2 F} = \frac{H}{4L}.$$

As a consequence, assuming that a single short vector is enough, the cost of a lattice attack is:

$$\text{either} \quad \frac{7.2\,L}{H} - 110 \quad \text{or} \quad \frac{0.144\,L^2}{H^2} - 27,$$

depending on the model we are using.

Since L/H is by construction a power of 2, the number of options is very restricted. We see that both models predict that $L/H = 16$ should be easy to attack. They also agree that $L/H = 32$ provides a security level of about 120 bits; possibly a bit more if we consider the number of repetitions necessary for the distinguishing attack and the fact that runtime is given in seconds. With $L/H = 64$, the estimates are respectively above 350 and 500; either of them should rule out all lattice attacks.

It should be noted that there is a complete family of estimates for the cost of lattice reduction depending on the type of algorithms and/or machine that are used. Many of those are covered in [APS15]. However, with the two simple models in agreement, it is reasonable to consider the choices $L/H = 32$ or $L/H = 64$ for cryptographic parameters.

4.4 Arora–Ge style algebraic attacks

In order to consider algebraic attacks in the style of [AG11] on our scheme, we need to express our knowledge about the key $\mathcal{S}$ and the noise in encryptions as algebraic equations. Of course, each of the L key bits satisfies $s_i^2 = s_i$. In addition, assuming that the noise level is at most ε, i.e., that every component of the noise vector belongs to $[-\varepsilon \cdots \varepsilon]$, we see that the components of the noise all satisfy the degree-$2\varepsilon + 1$ equation $\prod_{i=-\varepsilon}^{\varepsilon}(x - d) = 0$. Furthermore, every encryption of zero gives a linear relation between the bits of the key and the L chunks of noise in the encryption.

We now make the paranoid assumption that the attacker is able to write down every chunk of noise as a linear expression of the key bits. Note that we do not know how to achieve this goal in our system. With this assumption, every encryption of zero gives an extra degree $2\varepsilon + 1$ equation about the key bits. Moreover, every monomial in these equations is square-free, thanks to the $s_i^2 = s_i$ relations.

As a consequence, after collecting enough encryptions of zero, it is possibly to linearize the system of equations and solve a linear system in $\binom{L}{2\varepsilon+1}$ unknowns. Assume that we choose ε to be proportional to L, i.e., $\varepsilon = \epsilon L$, then asymptotically, since linear algebra is super-quadratic, the cost is more than $2^{2\mathcal{H}(2\epsilon)L}$ where $\mathcal{H}$ is the entropy function defined by $\mathcal{H}(x) = -x\log_2 x - (1-x)\log_2(1-x)$. With $\epsilon = 0.025$, this cost is already higher than the cost of combinatorial algorithms.

Note. It is easy to write linear relations involving a sum of the noise and the addition carry from the previous block using formulas similar to the ones used in the bit-extraction procedure. However, writing similar equations without the carries seems much more difficult. As a consequence, thanks to the presence of carries, it might possible to use noise components in the restricted range $\{-1, 0, 1\}$, without compromising the security against algebraic attacks. This is especially useful for the noise involved in the encryption of key-bits that are provided for the bit-extraction procedure. Since there is only a limited supply of such encryption, it seems that choosing $\varepsilon_{\text{key}} = 1$ can be a viable option, especially

for optimized parameters. Studying the exact cryptanalytic consequences of such a choice is an interesting topic for future research.

4.5 Unoptimized parameters

To design an unoptimized set of parameters, we choose them to satisfy the bounds that guarantee that the errors never become too large, in a way compatible with the security analysis. We consider the case $L/H = 64$. This leads us to the choice $H = 64$, $L = 4096$ and $\varepsilon_{\text{key}} = 128$. As a consequence, we get a very large Fermat number $2^{2^{18}} + 1$ and a slow system.

4.6 Optimized parameters

For optimization, we restrict ourselves to $L/H = 32$ and we rely on the carry propagation as protection against the Arora-Ge style attack, restricting the noise level when encrypting key-bits to $\varepsilon_{\text{key}} = 1$. For input data, a much larger noise level can be used.

Basically, two main parameters choices are possible. The first choice is $H = 16$ and $L = 512$, which leads to a quite small Fermat number $2^{2^{8192}} + 1$. Unfortunately, with the choice, the system is functional only using the first method for the bit-extraction of an `AND` gate. Furthermore, it is necessary to provide all encryptions $2^i s_j$ which makes the bit-extraction key quite large and also slows down the bit-extraction because of the number of multiplications which become necessary for each Mux.

The second choice is $H = 32$ and $L = 1024$ together with the Fermat number $2^{2^{65536}} + 1$. This is similar to the *TRGSW* parameters of [CGGI18]. Note that in contrast with this system, we prefer to provide a single parameter set; as consequence of that is that the key $\mathcal{S}$ contains twice as many bits which slows the computation by a factor of 2. With this choice, the system remains functional with second method for `AND` gates given only encryptions of $2^{16} s_j$ (and $-2^{16} s_j \mathcal{S}$) as bit-extraction key.

5 Test implementation

In order to test the system, we have implemented the computation of the minimum of two 16-bit numbers as already proposed on the THFE website tutorial found at https://tfhe.github.io/tfhe/coding.html.

We made all the experiments on a Intel Core i7–7700HQ at 2.8GHz. With THFE code, the timings are highly dependent on the FFT library which is used. With the fastest and very optimized library (Spqlios using FMA), the full homomorphic computation is done in 0.8 second. With the Nayuki FFT

library, the runtime is much higher, 7.6 seconds with the portable version and 3.3 seconds with the AVX version.

With the parameter set $H = 32$ and $L = 1024$, which is comparable to TFHE parameters, a pure GMP implementation of the arithmetic takes 16.2 seconds to perform the homomorphic computation of the minimum. With an approximate multiplication based on a portable implementation of the NTT (see the Appendix) the time is reduced to 9.4 seconds. As we mentioned above, since the key $\mathcal{S}$ has twice as many bits in our case, this slows our system down by a factor of two. Making the key sparser to reduce its size accordingly indeed reduces the time to 8.1 seconds with the GMP version and 4.7 seconds with the NTT version. We thus see that the results are comparable to the Nayuki version of TFHE.

In addition, let us mention the timings for the parameter set $H = 16$ and $L = 512$. They are respectively 19.3 seconds for the pure GMP implementation and 13.4 seconds for the NTT version. So despite the reduced size of the Fermat number, it is unfortunately much slower.

It remains to be seen how further our test implementation can be optimized by rewriting an approximate multiplication using a very fast FFT algorithm. However, it is likely than further effort, possibly with better parameter selection might considerably improve the performance of Fermat FHE.

6 Conclusion

In this paper, we propose an instantiation of FHE, inspired by existing systems but relying on the simple ring of integers modulo a Fermat number. The use of this simple ring allows to describe the system in more widely accessible terms and to hide many implementation details for the mathematical description of the system. Further study of the security and of advanced implementation techniques for this Fermat FHE system is necessary in order to further optimize it and to get a precise comparison with other systems.

Researching further variations allowing more functionalities is also a promising direction for the future. We provide additional information in the Appendix.

Acknowledgments

This work has been supported by the European Union's H2020 Programme under grant agreement number ERC-669891.

References

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011*, pages 403–415, 2011.

[AP14] Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, pages 297–314, 2014.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Mathematical Cryptology*, 9(3):169–203, 2015.

[CGGI16] Ilaria 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, 2016.

[CGGI17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In *Advances in Cryptology - ASIACRYPT 2017*, pages 377–408, 2017.

[CGGI18] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. Cryptology ePrint Archive, Report 2018/421, 2018. https://eprint.iacr.org/2018/421. Extended version of [CGGI16] and [CGGI17].

[DM15] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology - EUROCRYPT 2015*, pages 617–640, 2015.

[EHN+18] Keita Emura, Goichiro Hanaoka, Koji Nuida, Go Ohtake, Takahiro Matsuda, and Shota Yamada. Chosen ciphertext secure keyed-homomorphic public-key cryptosystems. *Des. Codes Cryptography*, 86(8):1623–1683, 2018.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009*, pages 169–178, 2009.

[LP11] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In *Topics in Cryptology - CT-RSA 2011*, pages 319–339, 2011.

[RAD78] Ronold L. Rivest, Len Adleman, and Michael L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, 4:169–180, 1978.

[vDGHV10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in Cryptology - EUROCRYPT 2010*, pages 24–43, 2010.

Possible variants and extensions

Approximate multiplication modulo F

In the description of the system, we use multiplication modulo F everywhere. However, a noisy multiplication which leads to invalid low-order bits in some blocks suffices. This can be easily achieved by multiplying polynomials modulo smaller power of two. For example, given two numbers $A = \sum_{i=0}^{L-1} a_i 2^{iH}$ and $B = \sum_{i=0}^{L-1} b_i 2^{iH}$, we can compute the product of $\mathcal{A}(X) = \sum_{i=0}^{L-1} a_i X^i$ and

$\mathcal{B}(X) = \sum_{i=0}^{L-1} b_i X^i$ modulo H^2 and modulo $X^L + 1$. If $\mathcal{C}(X)$ denotes the result of this multiplication, then $\mathcal{C}(2^H)$ is an approximation of $AB \pmod F$.

This multiplication of polynomials can be performed with fast NTT or FFT-based algorithms. Furthermore, the number of FFT/NTT computations can be limited by keeping encrypted key bits in their FFT form.

Key switching

We already know that the bit-extraction procedure can be used with distinct input and output keys and parameters. However, the output parameter L_{out} cannot be smaller than L_{in}. As a consequence, we need a way to move back from a larger parameter set to a smaller one. This can be done using a key switching similar to the one in [AP14]. Furthermore, this key-switching procedure cleans up the obtained ciphertext by zeroizing all blocks of decrypted value expect the low order one.

As in the bit-extraction procedure, we want to compute a reencryption of:

$$b_0 - a_0 s_0 + \sum_{i=1}^{L-1} a_i s_i$$

If we decompose each a_i in binary, we just need to add/subtract encryptions of $2^j s_i$ (with the new parameters) corresponding to bits set to 1. We can omit from the sum any input bit position that is outside of the output block-size.

For this key-switch procedure, the output block-size cannot be larger than the input block-size, otherwise, the high order bits in the decryption would be unpredictable.

Note that this key-switch procedure can also be used in a circular version to clean up the ciphertext keeping only the lowest order block and zeroizing the others. It is important to remark that this procedure is much faster than bit-extraction since it doesn't require the long sequence of Mux operations.

Ciphertext compression

In the textbook description of the system, every encryption is a pair (A, B) of numbers modulo F. However, during homomorphic computation only the low-order block of B is needed. Thus, ordinary ciphertext can be reduced by almost a factor of two by only keeping the low-order block B_0.

For the encrypted key bits used in the bit-extraction process, the situation is different. Indeed, the value of B is required to perform the Mux operation correctly. However, it is possible in that case (as mentioned in [CGGI18]) to compress the A part of the encrypted key by generating all the A-values that are needed from pseudo-random generator. They can thus be compressed by just giving out the seed of the generator. Again, this leads to a compression by almost 2.

Sparse or structured keys

In order to speed-up the bit-extraction, we need to limit the number of Mux that are required. Since each key-bit requires a Mux, this can be achieved by using sparser keys. For example, if we only have non-zero key-bits at even position, the number of Mux operations is halved.

Alternatively, we can structure the key to have a single bit set to 1 in an adjacent pair of key-bits at position $2i$ and $2i+1$. This also reduces the number of Mux to $L/2$ instead of L.

The impact of such sparse or structured keys on security needs to be studied deeper.

Other gates

In addition to the `AND` and `XOR` gates, arithmetic modulo 4 can easily be used to compute other binary gates. It is also possible to compute gates with more than two inputs. For example, given three bits x, y and z, $x + y + z \pmod 4$ is a two-bit number with the `XOR` of x, y and z as low-order bit and the majority as high order bit.

Other moduli

In addition to using Fermat numbers, the system can be adapted to pseudo-Fermat or pseudo-Mersenne number of the form $2^{HL} \pm 1$, where H is not necessarily a power of 2. It is preferable to keep a power of 2 for L since bit-extraction requires computation modulo powers of 2. However, if key-switching is used, this is not even necessary. Allowing these options would provide finer-grained parameters choices allowing to better tune the performance/security ratio.

Using an even wider variety of numbers such as $Q^L \pm 1$ might even be possible. However, the possible gains are less clear.