Fast Evaluation of Homomorphic Encryption Schemes Based on Ring-LWE

Cyrielle Feron, Vianney Lapotre, Loïc Lagadec

Fast Evaluation of Homomorphic Encryption Schemes based on Ring-LWE.

Cyrielle FERON
ENSTA Bretagne
UMR 6285, Lab-STICC
29806, Brest, France
cyrielle.feron@ensta-bretagne.org

Vianney LAPOTRE
Univ. Bretagne-Sud
UMR 6285, Lab-STICC
F-56100 Lorient, France
vianney.lapotre@univ-ubs.fr

Loïc LAGADEC
ENSTA Bretagne
UMR 6285, Lab-STICC
29806, Brest, France
loic.lagadec@ensta-bretagne.fr

***Abstract*—When evaluating Homomorphic Encryption (HE) schemes, only one set of input parameters is usually considered. Evaluation reports HE scheme time execution and memory consumption because these are the main challenges of HE. PAnTHErS enables to evaluate HE schemes without executing the scheme, hence with an affordable processing time. Results are provided in terms of computational complexity and memory cost. This allows to evaluate a scheme for numerous sets of input parameters. In this paper, PAnTHErS is improved by a calibration phase, and four HE schemes based on Ring-LWE are analyzed and compared using the proposed tool.**

***Index Terms*—Homomorphic Encryption, Security, Cloud Computing**

I. INTRODUCTION

Contrary to classical cryptosystems, Homomorphic Encryption (HE) allows to directly apply computation on ciphertext. However, as HE schemes are many, electing one HE scheme and its associated parameters for a given application remains challenging. Indeed, since Gentry [1] designed the first Fully HE (FHE) scheme in 2009, lots of HE and FHE schemes have been created. Although the first scheme was based on ideal lattices, the following ones relied on different hardness assumptions (approximated-GCD, Learning With Error (LWE), Ring-LWE and approximate-eigenvector). Since then, several implementations of HE schemes have been made available in open-source. Among them, HElib [2] and FV-NFLlib implement Ring-LWE based schemes [3] and [4] respectively. However, existing implementations exhibit a too high execution time to support exploring all HE schemes then selecting one for a given application.

HE schemes suffer from both an important complexity and a high memory consumption. As a result, real applications exploiting HE schemes are quite few. These two flaws are being reduced over time thanks to new technologies and development of new optimized schemes. Nevertheless, finding the best HE scheme for a particular application remains difficult. Evaluation requires the application to be run on various sets of input parameters for every each tested scheme. In this context, HE schemes exploration could help choosing the most interesting scheme and configuration corresponding to application requirements.

PAnTHErS (Prototyping and Analysis Tool for HE Schemes) [5] allows HE experts to analyze HE schemes in terms of computational complexity and memory cost without executing the scheme itself. The tool is based on an analytic evaluation of HE schemes allowing fast exploration. However, PAnTHErS produces a theoretical complexity (number of operations) and a theoretical memory cost which are not related to a specific implementation.

In this work, we propose to extend PAnTHErS by integrating a calibration phase. This would allow to ensure the theoretical results match to practical results which fit a particular system and a given implementation. The proposed approach is validated through implementing and analyzing four HE schemes based on Ring-LWE ([4], [6], [7], [8]). Then, schemes are evaluated on a synthetic HE application with an arbitrary number of homomorphic multiplications and additions.

This paper is organized as follows. Section II presents PAnTHErS. Section III focuses on the calibration phase and evaluates it against four HE schemes. Then, Section IV shows evaluation results on a case study. Finally, Section V concludes and presents future works.

II. PANTHERS PRESENTATION

PAnTHErS [5] is a tool implemented in Python. It provides a modeling phase to decompose complex HE schemes into series of simpler mathematical functions. Each created function is stored in a library for further reuse in other decompositions. PAnTHErS models can be analyzed in terms of computational complexity and memory cost with no need for executing the HE scheme itself.

This section recaps modeling and analysis steps of PAnTHErS. These steps are fully detailed in [5]. Moreover, usage of PAnTHErS is also explained.

A. Modeling step

First, HE schemes to be analyzed, must be modeled into series of functions. PAnTHErS considers three kind of functions: atomic, specific and HE basic.

An atomic function corresponds to one operation. Specific and HE basic functions are series of atomic and/or specific functions. Each HE scheme is composed of five functions: Key Generation, Encryption, Addition, Multiplication and Decryption. These functions are modeled as HE basic functions.

Atomic and specific functions are stored in a library to favor reuse, whichever other schemes which are based on the same hardness. This approach speeds up the modeling step of new HE schemes.

B. Analysis step

Each HE scheme is analyzed in terms of computational complexity and memory cost. In PAnTHErS, complexity is represented by a table containing the number of operations performed for a given application scenario. The operations being counted in PAnTHErS are: multiplication, addition, division, subtraction, modulo, random and round.

Memory cost, also summed up in a table, is composed of the characteristics of parameters. The table itemizes output parameters as well as temporary variables. The characteristics stored in PAnTHErS are: type (integer or polynomial), dimensions (number of rows and columns) and degree (if the parameter is a polynomial).

Analysis requires on two functions: one for complexity and one for memory cost. Each atomic, specific and HE basic function is associated to its own two functions. Once the execution of analysis function is completed, PAnTHErS returns: total complexity and total, maximal and current memory cost. Since these results are obtained at the function level, it is possible to produce temporal traces for each metric.

Thanks to a method detailed in [5], PAnTHErS transforms the complexity table into a total complexity: an integer corresponding to the number of multiplications performed on integers during the HE scheme. Total memory cost covers all parameters which have been created during HE scheme analysis execution. Total memory is defined by an integer value representing the number of stored 32-bit integers. Current memory cost refers to parameters being stored in memory at this exact moment. PAnTHErS also returns the maximal memory cost achieved during the analysis execution.

C. Usage for HE schemes exploration

In order to use HE in an application, designers look for the scheme that best fits the application requirements. Also, the most interesting input parameters must be isolated. The execution of the application with various schemes and numerous sets of input parameters results in important time and memory costs. PAnTHErS alleviates the need for this costly step.

First, if the necessary models are not available in the library, the designer provides descriptions of specific functions, then descriptions of HE basic functions to model a HE scheme. Once modeled a HE scheme, a set of input parameters can be provided for the exploration process. Each input parameter is associated to a range of values and a step. PAnTHErS returns one analysis per input set. Results are returned in the form of graphs showing the evolution of one parameter at a time.

Then, based on graphs and the application requirements, the most favorable scheme and/or input parameters are designated.

III. PROPOSED CALIBRATION PHASE

PAnTHErS supports fast analysis of HE schemes by returning their theoretical complexity and memory consumption. On the other side, the evaluation of one HE scheme, through PAnTHErS returns the evolution of both complexity and memory cost according to each input parameter. However, how a scheme executes strongly depends on both its implementation and the target architecture. For instance, a C implementation and a Python implementation exhibit differences.

In this section, we propose a calibration phase included in PAnTHErS in order to provide sound results regarding a given implementation. Then, the calibration is applied for the evaluation of four HE schemes based on Ring-LWE and finally, it is evaluated.

A. Calibration method

The calibration method aims to transform theoretical computational complexity and memory cost, that PAnTHErS returns, into a practical case for a given implementation.

For each HE scheme, PAnTHErS gives an integer representing the complexity and an integer corresponding to the memory cost. In this paper, we call *CompPanthers* (resp. *MemPanthers*) the complexity (resp. memory cost) returned by PAnTHErS. We propose to actually execute the HE scheme in order to get its execution time in seconds for a practical case (*e.g.* a Python implementation in this paper). Moreover, we apply a memory profiler (Python module named *memory_profiler*[1] in this paper) on each HE scheme execution. The module returns memory consumption of the program in Mebibytes (MiB). Thus, in the following, complexity in the practical case, named *CompPractical*, refers to Python HE scheme execution time in seconds. Memory cost in the practical case, named *MemPractical*, refers to Python HE scheme memory cost calculated by *memory_profiler* in MiB.

The calibration method associates *CompPanthers* with *CompPractical* and *MemPanthers* with *MemPractical*. Thus, it creates two points (*CompPanthers*, *CompPractical*) and (*MemPanthers*, *MemPractical*). Each point enables to transform a PAnTHErS analysis into a practical analysis. Indeed, for complexity, thanks to the association of *CompPanthers* with *CompPractical*, it is possible to find a factor y such as:

$$CompPanthers \times y = CompPractical.$$

Factor named y allows then to convert any analysis of PAnTHErS into practical analysis. The same reasoning is valid for memory cost.

In order to improve the proposed approach, two complexities (resp. memory costs) could be associated to two execution times (resp. MiB) creating thus two points. With these points, two factors y and z can be recovered to solve the following system:

$$\begin{cases} CompPanthers_1 \times y + z &= CompPractical_1 \\ CompPanthers_2 \times y + z &= CompPractical_2 \end{cases} \quad (1)$$

Then, (y,z) are applied to any other analysis of PAnTHErS complexity. Calibration method with two points could give a better approximation of PAnTHErS values in a practical

[1] https://pypi.python.org/pypi/memory_profiler

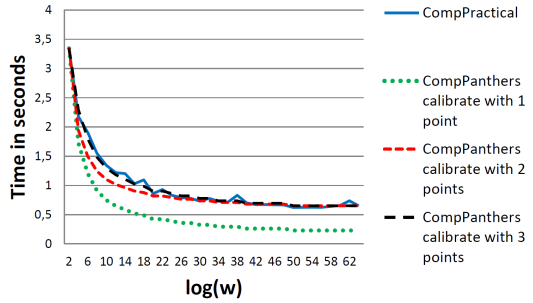


Figure 1: Comparison between execution time of FV scheme (by varying $\log_2(w)$) and complexity calibrated with 1 to 3 points. Fixed parameters: $\log_2(q) = 200$, $t = 2$ and $n = 2^{11}$.

case than the calibration method with one point. We called a p-calibration, a calibration with p points.

Figure 1 shows results of calibration method on FV scheme [4]. In FV, operations manipulate polynomials of degree n and are reduced modulo q. Parameter t is the modulus of plaintexts and w is an integer base. Figure 1 illustrates execution time while parameter w evolved and q, t and n are fixed. We varied $\log_2(w)$ from 2 to 64 with a step of 2. FV was executed for each $\log_2(w)$ to find the *CompPractical* values. The three other curves illustrate calibrations of *CompPanthers* values according to the number of associations (*i.e.* points).

The comparison of curves in Figure 1 shows that the curve corresponding to a 3-calibration is closer to practical values. The percentage of error between *CompPractical* and the three *CompPanthers* as given by:

$$\frac{|CompPractical - CompPanthers|}{CompPractical} \times 100. \quad (2)$$

Maximal error for a 1-calibration goes to almost 70% while it goes only to 12% for a 3-calibration. For a 2-calibration, maximal error goes to 22%.

Migrating from p to the p-calibration method is likely to decrease the error between practical and calibrated PAnTHErS values. However, a point comes from executing at least one time the HE scheme. This execution takes a time denoted *HEtime*, expressed in seconds. If the calibration is done with p points, HE scheme has to be executed at least p times. Thus, it requires $p \times HEtime$ seconds before using calibration method and obtaining results. Therefore, a tradeoff must be done between execution time and percentage of error to adjust PAnTHErS values in a practical case.

B. Calibration validation

Four schemes have been modeled and analyzed in PAnTHErS to study the efficiency of the proposed calibration phase. For that purpose, a practical Python implementation executed on an Intel(R) Core(TM) i5-4310M CPU 2.70 GHz machine has been considered. Moreover, we use a 2-calibration as it seems to be an interesting tradeoff between execution time and the percentage of error.

The four schemes named FV [4], YASHE [6], F-NTRU [7] and SHIELD [8] are defined in the ring $R_q = R/qR$ where

Table I: Parameters variation of each scheme for analyses.

Scheme	Varied parameters				Fixed parameters			
	Name	Min	Max	Step	$\log_2(q)$	n	w	t
FV and YASHE	$\log_2(q)$	100	300	10	-	2^{12}	2^2	2
	$\log_2(w)$	2	42	2	100	2^{12}	-	2
	t	2	60	1	100	2^{12}	2^2	-
SHIELD	$\log_2(q)$	4	40	2	-	2^{10}	-	-
F-NTRU	$\log_2(q)$	4	40	2	-	2^{10}	2^2	-
	$\log_2(w)$	1	16	1	20	2^{10}	-	-

Table II: Times required for calibration phase versus execution times of each parameter of each schemes.

Scheme	Calibration time	Execution time	Speedup
FV	32.2"	7'25"	13.8
YASHE	2'46"	41'04"	14.8
SHIELD	58.3"	6'30"	6.7
F-NTRU	37"	2'27"	4.0
All	4'54"	57'26'	11.7

$R = \mathbb{Z}[x]/(\phi_d(x))$, $\phi_d(x)$ is the irreducible dth cyclotomic polynomial. Operations are made on polynomials of degree $n = \phi(d)$. Plaintexts are reduced modulo t in FV and YASHE. And, some words are decomposed in base w in FV, YASHE and F-NTRU schemes.

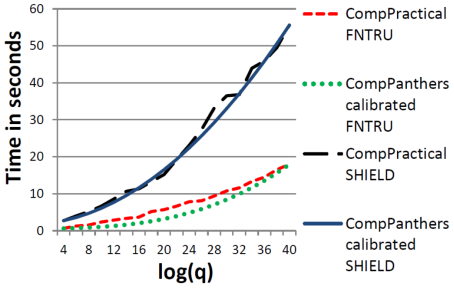
The evaluation of calibration phase was made varying the following parameters: q, w and t. Table I recaps the variation of each parameter executed for PAnTHErS analyses. For the 2-calibration, we decide to associate PAnTHErS complexity (resp. memory cost) with execution time (resp. MiB) at the first and last value of each range. For instance, the 2-calibration of q graph for FV scheme complexity have been done taking PAnTHErS complexity and Python execution time at $\log_2(q) = 100$ and $\log_2(q) = 300$. In addition, two more 2-calibrations are produced for w and t for the same scheme. At last, three 2-calibrations are necessary for FV and YASHE, two 2-calibrations for F-NTRU and only one for SHIELD. So, twelve HE scheme executions are required to composed points for the six 2-calibrations.

Considering parameters of Table I, times required to ensure 2-calibration for each scheme are exposed in Table II. The third column shows the total execution time of the considered HE schemes when exploration is done through actual execution. Results show that the proposed approach allows to calibrate PAnTHErS while analysis speedup is still significant. It is worth noting that speedup increases with larger sets of parameters, meaning that this approach is highly scalable.

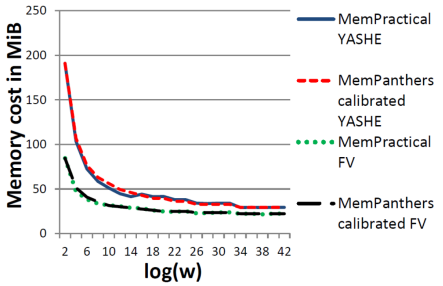
We evaluated the calibration method on each HE basic and on each whole HE scheme. Some of HE basic functions have an execution time (resp. memory cost) below 0.1 second (resp. MiB). As these orders of magnitude are tiny, they are negligible. Thus, results of these HE basic functions were not included into general results of this paper.

Figures 2a and 2b illustrate the evolution of practical values next to calibrated PAnTHErS values. In graphs of Figure 2, PAnTHErS values are close to practical values: their evolution are equivalent.

Figure 3 gives box plots created with mean percentages of



(a) Total F-NTRU and SHIELD complexity results (with q parameter).



(b) Total FV and YASHE memory cost results (with w parameter).

Figure 2: Evolution of practical values and calibrated PAnTHErS values.

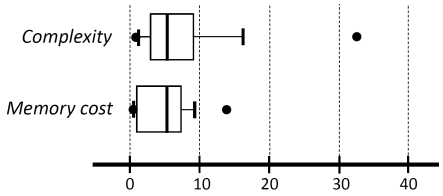


Figure 3: Box plots showing the dispersion of percentages of mean error.

error. These percentages are the ones found after analyzing each HE basic functions of the four schemes. We took only HE basic functions with an order of magnitude bigger than 1. For computational complexity and memory cost, three quarters of the percentages of mean error are under 9%.

Table III gives percentages of error for HE schemes. Mean percentage for complexity are under 5.6% expect for F-NTRU. These means signify a difference of maximal 1 second between *CompPractical* and *CompPanthers*. On Figure 2a, difference between *CompPractical FNTRU* and *CompPanthers calibrated FNTRU* goes up to 55.7% according to Table III. Differences between the two curves are less than 3 seconds.

The p-calibration phase enables to transform the theoretical analyses of PAnTHErS into practical results: complexity (resp. memory cost) of PAnTHErS is changed in seconds (resp. MiB) thanks to Python executions for our study. A p-calibration associates p PAnTHErS results with p practical-case results found by executing a HE scheme. An important p implies more significant pre-calculations. We took $p = 2$ to evaluate the calibration phase with four HE scheme based on Ring-LWE. The majority of percentages of error are under 20% for

Table III: Mean and maximal percentages of error for each HE scheme and parameter.

Scheme	Complexity		Memory cost	
	Mean	Max.	Mean	Max.
FV	3.9%	16.5%	3.2%	14.3%
YASHE	5.6%	39.5%	3.7%	11.7%
SHIELD	3.5%	11.7%	6.4%	17.2%
F-NTRU	24.6%	55.7%	6.9%	20.1%

complexity and under 15% for memory consumption. Theoretical PAnTHErS results and practical-case results follow similar trends. Next section focus on apply PAnTHErS and evaluate calibration phase on a case.

IV. CASE STUDY

In order to demonstrate the interest of the proposed tool, we consider in this section a synthetic application. Using PAnTHErS, the aim is to find the best HE scheme among FV, YASHE and F-NTRU for the application and the best configuration of the HE scheme chosen targeting a Python on an Intel(R) Core(TM) i5-4310M CPU.

The application was created with an arbitrary number of HE basic functions. Multiplicative depth is the number of homomorphic multiplications which can be executed in succession. In literature, existing and practical applications using HE have a multiplicative depth between 8 and 12 ([9], [10]). We wanted that our application has a similar depth. Consequently, we create an application with a depth of 10. The application begins with a key generation and then, three encryptions are performed. Messages m_1, m_2 and m_3 are encrypted into c_1, c_2 and c_3. The application then does the following calculations:

$$c_1^3 c_2^2 c_3^2 ST(S+c_3)(S+c_2+c_3) \tag{3}$$

with $S = c_1T + c_1 + c_2$ and $T = c_1c_2 + c_3$. Finally, the calculation result is decrypted.

The application was analyzed by PAnTHErS with FV, YASHE and F-NTRU. Results of PAnTHErS lead to determining the best HE scheme for the application. Simultaneously, we executed each Python implementation for the sake of verifying the efficiency of both PAnTHErS and the calibrated phase.

Input parameters must be chosen in order to ensure the accuracy of algorithm results. In other words, input parameters must ensure that the HE scheme can do its ten successive homomorphic multiplications. Equations given in [4], [6] and [7] enable to calculate the depth according to the input parameters. [8] does not present the whole depth calculation. Thus, SHIELD was not tested on the application.

First, we analyzed the application by varying parameter q. For FV and YASHE, we took $\log_2(q) \in \{300, 305, ..., 350\}$ and fixed $w = 2^2$, $t = 2$ and $n = 2^{12}$. For F-NTRU, we took $\log_2(q) \in \{90, 95, ..., 140\}$ and fixed $w = 2$ and $n = 2^{10}$.

As in Section III, we used a 2-calibration by associating PAnTHErS complexity (resp. memory cost) to Python execution time (resp. MiB) at the first and the last value of $\log_2(q)$ range. The analysis of parameter q for the 3 schemes requires about 120 minutes while it would take around 540 minutes for

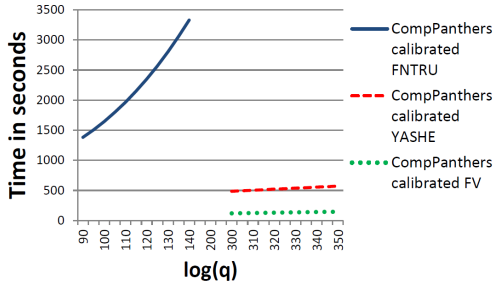


Figure 4: Calibrated complexity evolution of FV, F-NTRU and YASHE in function of $\log_2(q)$. Fixed parameters: $w = 2^2$, $t = 2$ and $n = 2^{12}$.

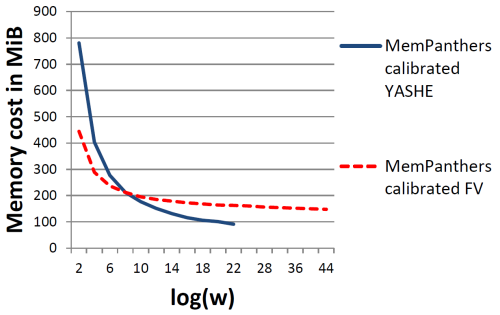


Figure 5: Calibrated memory cost evolution of FV and YASHE in function of $\log_2(w)$. Fixed parameters: $\log_2(q) = 300$, $t = 2$ and $n = 2^{12}$.

executing each HE scheme with each $\log_2(q)$. A 3-calibration would imply an analysis of 149 minutes.

Figure 4 shows calibrated complexity evolution of the application with each scheme. Results for memory cost are very close from the ones for complexity. F-NTRU curve is higher and increases faster than FV and YASHE curves. This first analysis enables us to remove F-NTRU and resume analyses only with FV and YASHE. Before validating our choice to continue only with FV and YASHE, we verified the quality of our results. The maximal percentage of error went only to 6.5%. PAnTHErS with its 2-calibration was efficient and allowed us to make a good choice. The usage of a 3-calibration implies a maximal percentage of error of 3.9%.

For next analysis, we varied $\log_2(w) \in \{2,4,...,22\}$ for YASHE and $\log_2(w)$ went to 44 for FV. Indeed, a $\log_2(w) > 22$ with $\log_2(q) = 300$ implies a depth inferior to 10 for YASHE. For FV, depth stays at 10 until $\log_2(w) = 44$. Evolution of complexity in function of $\log_2(w)$ shows that FV is the most interesting because it always has a complexity inferior to the one of YASHE. However, Figure 5 shows that for $w > 2^8$, YASHE consumes less memory than FV. If there are memory constraints in application requirements, then YASHE seems to be the best choice. Here, FV is the most interesting according to complexity results.

The last analysis focused on variation of $t \in \{2,3,...,9\}$ for FV scheme. As in Section III, t does not has an impact on computational complexity and memory cost of the application.

Python execution of all sets of parameter leads to same results as PAnTHErS predicted. PAnTHErS analyses showed FV scheme was the best candidate for the application. An appropriate configuration is to take a small q. Indeed, computational complexity increases when q grows. Moreover, an important w reduces memory cost and complexity. From $\log_2(w) = 28$, evolution of complexity and memory cost stagnate (see Figure 5). In addition, depth decreases when w increases. Thus, the value 2^{28} for w is a reasonable choice. Any t is convenient as long as it does not reduce the depth. Thus, $t = 2$ is suitable for FV configuration. As $w = 2^{28}$ and $t = 2$, we found the smallest q which implies a depth of 10. Thus, $\log_2(q) = 281$. Finally, n depends on q. If $\log_2(q) = 281$, then n must have 2^{12} as a value to ensure 80-bit security [11].

V. CONCLUSION AND FUTURE WORK

Prior to this work, PAnTHErS supported theoretical evaluation of HE schemes in terms of computational complexity and memory consumption. This paper describes a p-calibration phase that changes these theoretical values into practical values matching a given system and a given implementation. The p-calibration method was evaluated on four schemes based on Ring-LWE and an application. Higher value of p is, the more considerable pre-calculations are, and better PAnTHErS calibrated results are.

This open exciting perspective for future work, such as analyze practical applications with various HE schemes based on different hardness assumption. These analyses could target different systems and/or implementations. Based on these analyses, HE schemes and its applications could be executed in the best conditions *i.e.* choosing the environment that favors the fastest execution time.

REFERENCES

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices." in *STOC*, 2009, pp. 169–178.
[2] S. Halevi and V. Shoup, "HElib - An implementation of homomorphic encryption," https://github.com/shaih/HElib, 2014.
[3] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) Fully Homomorphic Encryption without Bootstrapping," in *ITCS*, 2012.
[4] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.
[5] C. Feron, V. Lapotre, and L. Lagadec, "PAnTHErS: A Prototyping and Analysis Tool for Homomorphic Encryption Schemes," in *SECRYPT*, 2017.
[6] J. W. Bos, K. E. Lauter, J. Loftus, and M. Naehrig, "Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme," in *Cryptography and Coding IMA*, 2013.
[7] Y. Doröz and B. Sunar, "Flattening NTRU for evaluation key free homomorphic encryption," *IACR Cryptology ePrint Archive*, 2016.
[8] A. Khedr, P. G. Gulak, and V. Vaikuntanathan, "SHIELD: Scalable Homomorphic Implementation of Encrypted Data-Classifiers," *IEEE Trans. Computers*, 2016.
[9] A. Canteaut, S. Carpov, C. Fontaine, T. Lepoint, M. Naya-Plasencia, P. Paillier, and R. Sirdey, "Stream ciphers: A Practical Solution for Efficient Homomorphic-Ciphertext Compression," Cryptology ePrint Archive, Report 2015/113, 2015.
[10] S. Carpov, T. H. Nguyen, R. Sirdey, G. Costantino, and F. Martinelli, "Practical Privacy-Preserving Medical Diagnosis Using Homomorphic Encryption," in *CLOUD*, 2016.
[11] V. Migliore, G. Bonnoron, and C. Fontaine, "Determination and exploration of practical parameters for the latest Somewhat Homomorphic Encryption (SHE) schemes," *Working paper or preprint*, 2017.