Optimizing Homomorphic Encryption Parameters for Arbitrary Applications

Charles Gouert, Rishi Khan, and Nektarios Georgios Tsoutsos

Abstract

Homomorphic encryption is a powerful privacy-preserving technology that is notoriously difficult to configure, even for experts. In this article, we outline methodologies for determining optimal cryptographic parameters for any arbitrary application. We provide guidelines for both leveled and fully homomorphic encryption, and demonstrate the presented strategies with the BGV cryptosystem.

Index Terms

Encrypted computing, Homomorphic encryption, Parameter optimization.

I. Introduction

Fully homomorphic encryption (FHE) has seen increasing attention in the research community and industry as new cryptographic schemes and application-specific optimizations have demonstrated that FHE is fast approaching feasibility in realistic scenarios. This powerful class of encryption allows users to execute any arbitrary algorithm on encrypted data and serves as a robust solution to privacy concerns in the context of cloud computing. Users can encrypt sensitive data locally, upload the resulting ciphertexts to the cloud, instruct the cloud to execute an algorithm on the ciphertexts, receive the encrypted output, and decrypt locally to view the correct plaintext result of the computation. In this way, organizations can outsource computationally expensive calculations to the cloud while having strong data privacy guarantees.

While standard encryption mechanisms such as AES have been used to enforce data confidentiality for outsourced data in the past, these solutions only work for storage and transmission. If any meaningful computation needs to be done on the data, the user needs to pull the data from the cloud, decrypt, perform the computation on the plaintext data, re-encrypt, and finally re-upload to the cloud. Conversely, FHE

C. Gouert and N. G. Tsoutsos are with the Department of Electrical and Computer Engineering, University of Delaware, Newark, DE. E-mail: {cgouert, tsoutsos}@udel.edu. Rishi Khan is with Extreme Scale Solutions, New Castle, DE. Email: rishi@extreme-scale.com.

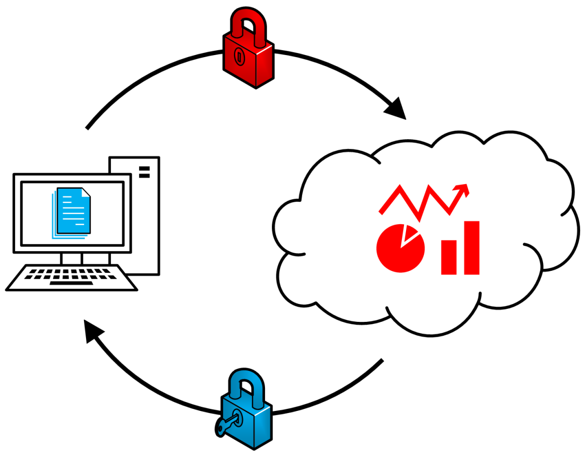

Fig. 1. **Secure Outsourced Computation:** A user has sensitive plaintext data (in blue), encrypts the data, sends the ciphertext (in red) to a cloud server that computes analytics, and returns encrypted results to the user, which the user decrypts locally.

can accomplish the same task since the computation can be done directly on the encrypted data stored on the cloud without ever involving the user. Because the *data in use* is encrypted, the cloud has no knowledge of the underlying plaintext. In addition, if a cloud server is compromised, the attacker will be unable to decrypt the homomorphic ciphertexts without the secret key. A diagram depicting this secure cloud computing scenario is shown in Figure 1.

Only a few years ago, the idea that FHE could be adopted in a cloud computing context outside of research circles was a distant dream. When it was first conceived in 2009 by Craig Gentry [1], it was estimated that the cost of performing a Google search over encrypted data would be a trillion times slower than a regular Google search [2]. Now, state-of-the-art FHE libraries [3] and novel computational techniques tailored for encrypted computation [4] have accelerated the evaluation time of homomorphic algorithms by several orders of magnitude. Further, leveled homomorphic encryption (LHE) provides an efficient alternative to FHE and can achieve much faster speeds for certain types of applications.

Unfortunately, while efficiency of both FHE and LHE operations continue to increase, usability is not advancing at the same speed. For instance, choosing optimal parameter sets at an acceptable security level for most HE libraries remains a notoriously difficult problem. While one set of parameters may result in efficient computation for one application, the same set of parameters may result in poor performance for a different set of computations on encrypted data. Even for crypto-savvy programmers, the process of optimizing encryption parameters is time consuming and requires a high degree of trial-and-error.

Prior works have identified solid and versatile parameter sets and existing HE libraries typically provide a set of default parameters, but these parameter sets are not tailored for any given application. In addition, the level of security may not be appropriate for some users as many parameter sets are preconfigured for 80 bits of security or less, which is considered fairly weak. For HE schemes such as BGV [5], default parameter sets may not be well suited for high degrees of *batching*. The latter is a very powerful technique applicable to a subset of HE schemes that allows multiple plaintext values to be encoded in a single ciphertext. When an operation is conducted with a batched ciphertext, each individual plaintext element will be affected, in a similar way to SIMD-style computation. For many types of applications such as neural network training and inference, this strategy can drastically improve throughput. The size of the plaintext vector that can be encoded is solely determined by the parameter set and can vary from less than 10 to several thousand. In all, selecting parameters well suited for the problem, one will often achieve speedups of multiple orders of magnitude compared to ill-suited configurations.

To address the problem of selecting good parameters for any given application, this article presents guidelines for selection gleaned from the authors' myriad experiences working with both leveled and fully homomorphic encryption to allow users to more easily wield the full power that current state-of-the-art HE libraries provide. The next section will give a high-level overview of homomorphic encryption and its different variants, as well as a brief look at some of the most important HE schemes. Then, we will proceed to identify methodologies for optimized parameter tuning for leveled homomorphic encryption followed by fully homomorphic encryption. Finally, we will use the BGV cryptosystem as a case study to demonstrate the presented strategies and derive efficient parameter sets for a variety of use-cases.

II. Homomorphic Encryption Primer

Homomorphic encryption comes in three distinct forms that each have their own set of pros and cons. The weakest, and oldest, form of homomorphic encryption is *partially homomorphic encryption* (PHE). PHE cryptosystems allow for an unlimited number of *either* multiplication or addition on ciphertext data. Popular schemes include asymmetric algorithms like RSA [6], which allows for multiplication between ciphertext values, and Paillier [7], where homomorphic addition is possible. This class of HE has a distinct advantage over the stronger classes of HE in that the ciphertexts are not *noisy* and no noise management techniques are necessary. Considering that noise mitigation techniques such as bootstrapping (discussed later in the section) form the bottleneck of HE computation, PHE can execute algorithms on encrypted data faster than the other two classes. However, the primary downside here is that solely addition or multiplication is not functionally complete, meaning that it is not possible to implement any arbitrary algorithm with only one type of arithmetic operation. Indeed, only relatively simple algorithms, such as aggregation, are possible with PHE techniques.

Leveled HE (LHE) is the next step up from PHE in terms of computational ability. Instead of supporting only a single type of arithmetic operation, LHE is capable of supporting both addition *and* multiplication

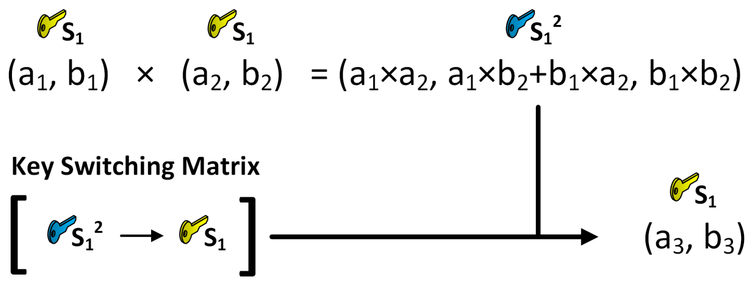
s_1
s_1
s_1^2
$(a_1, b_1) \times (a_2, b_2) = (a_1 \times a_2, a_1 \times b_2 + b_1 \times a_2, b_1 \times b_2)$
Key Switching Matrix
$[s_1^2 \rightarrow s_1]$
s_1
(a_3, b_3)

Fig. 2. **Key Switching:** HE multiplication results in a larger product ciphertext that is encrypted under the square of the secret key. Key switching serves to reduce the dimensionality of the product as well as remap the ciphertext back to a secret key.

with ciphertext data. Therefore, this class of HE is considered functionally complete and can be used to implement any algorithm, with one exception: The algorithm must have a *bounded depth*. LHE schemes, unlike standard PHE algorithms, derive their security from variants of the *learning with errors* problem [8]. Effectively, this means that when a plaintext is encrypted, a small amount of noise is injected to make the encryption harder to break without knowledge of the secret key. An unfortunate side effect is that as ciphertexts are used in more and more computations, the noise grows. When two ciphertexts are added, the noise grows linearly, while multiplication causes the noise to increase exponentially. If the noise exceeds a certain threshold, it will no longer be decryptable and is effectively rendered useless. If the depth of the HE arithmetic circuit, typically determined by the number of multiplications performed successively on ciphertext data, is bounded, it is possible to choose parameter sets that will allow the LHE scheme to successfully evaluate the circuit without exceeding the noise threshold.

The three most prevalent LHE schemes include BFV [9], BGV [5], and CKKS [10]. The first two encrypt integers modulo a user-defined plaintext modulus while the latter encodes floating point values. In these schemes, a ciphertext is encoded as a tuple of high-degree polynomials with large coefficients over a predefined ring. The size of the coefficients is constrained by a ciphertext modulus q, which is a product of primes. Each ciphertext polynomial is defined modulo an *irreducible cyclotomic polynomial* of order m, which effectively limits the maximum polynomial degree. In practice, m defines the number of coefficients present in each polynomial and q defines the sizes of each coefficient. As a result, the size of the ciphertexts scales with both q and m.

These LHE schemes also support batching, which allows for vector-like processing that can vastly improve the throughput of homomorphic operations since you can evaluate many multiplications or additions for the price of a single operation. If chosen carefully, parameter sets can allow a single ciphertext to hold potentially thousands of independent plaintext elements.

The simplest operation on ciphertext data is addition, which just becomes an addition of polynomials.

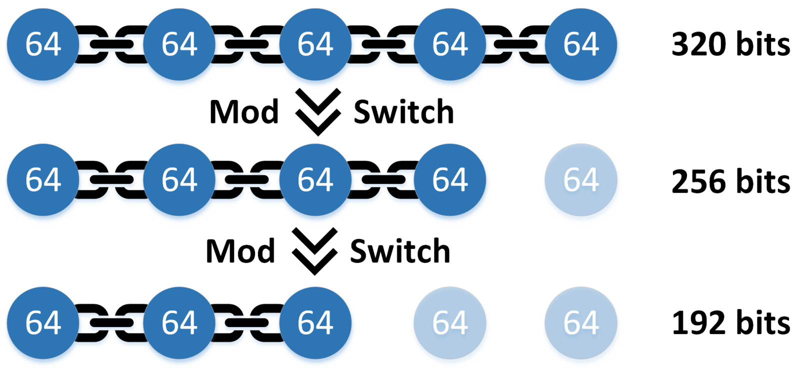

Fig. 3. **Modulus Switching:** This operation manipulates the chain of primes (of size 64 bits in this figure) composing the ciphertext modulus. Each invocation removes a prime from the chain and hence decreases the bit size of all coefficients in the ciphertext.

Conversely, multiplication is more expensive in terms of both execution time and noise accumulation. Multiplication involves multiplying the pairs of polynomials together, which results in the unwanted side effect of introducing a third large polynomial to the ciphertext tuple. Consequently, each multiplication will cause the ciphertext to become larger and larger and take up increasing amounts of memory. Luckily, there is a mechanism to deal with this and effectively constrain the ciphertexts to their original sizes: *key switching*. The name of this operation is derived from the fact that the product ciphertext is actually encrypted under the *square of the secret key*. This must be switched back to the original secret key, or a different secret key to avoid assuming circular security. Key switching will output a ciphertext with only two polynomials and more noise. This operation after a multiplication is illustrated in Figure 2.

A modulus switching operation (shown in Figure 3) will always follow a multiplication (and subsequent key switch). This operation solely affects the ciphertext modulus q, which can be thought of as a chain of prime numbers. By removing one of the primes from the chain, the ciphertext modulus will correspondingly decrease by the bit size of the removed prime. This effectively scales down the magnitude of the ciphertext coefficients, which also serves to scale down the magnitude of the noise. However, because there are a finite number of primes in the chain, this operation can only be done a limited number of times.

Even with the advantage of batching and the ability to mod switch, LHE becomes infeasible for complex algorithms as the ciphertext sizes will scale with the number of multiplications required to evaluate it. For these algorithms with a large depth, or algorithms with an unbounded number of operations, it is more appropriate to use the next, most powerful form of HE.

Fully homomorphic encryption (FHE) is identical to LHE besides the inclusion of the most effective

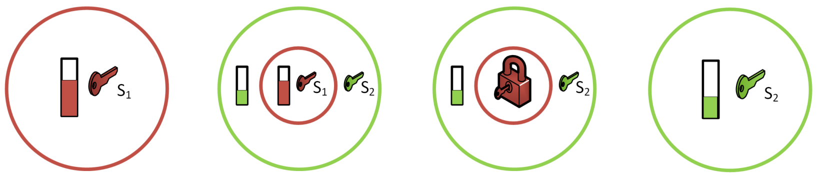

Fig. 4. **Bootstrapping:** Starting with a noisy ciphertext under secret key S_1, represented by a red circle, bootstrapping will add another layer of encryption by encrypting the ciphertext under key S_2 with less noise. Next, the inner layer of encryption is stripped away by executing the decryption function homomorphically. The result is a ciphertext under key S_2 encrypting the same message with reduced noise.

(and expensive) form of noise reduction: *bootstrapping*. Consider the following scenario in an LHE context: the cloud computes on a ciphertext until it becomes noisy and then asks the user to refresh the noise. The user can do this by downloading the ciphertext, decrypting it, and then sending the cloud a fresh encryption of the underlying plaintext value in the original ciphertext. Now, the cloud can proceed to do more operations with the ciphertext. However, this is a suboptimal solution as it requires the user to be active during the computation and increases the communication overhead associated with sending a large ciphertext from the cloud server to the user and vice versa. Bootstrapping takes the intuition from this scenario and allows it to happen entirely on the cloud server without any interaction from the user.

Bootstrapping effectively performs a decryption procedure *in the encrypted domain*, as is illustrated in Figure 4. In order to accomplish this, the user sends the cloud an encryption of the secret key, referred to as the evaluation key. One can envision this as happening inside another layer of encryption (i.e., the green circle), so the plaintext value is never exposed. The resulting ciphertext encodes the same underlying value as the noisy ciphertext, but the noise magnitude is reduced to a level that allows for more HE operations to be executed. It is important to consider the depth of the decryption circuit, which adds noise before it can be reduced, and thus requires some remaining noise budget to work correctly. Additionally, the complexity of the decryption circuit will affect the amount of noise capable of being removed by the bootstrapping procedure. Generally, the larger the plaintext modulus, the deeper the decryption circuit becomes.

Unlike other noise reducing techniques used by LHE schemes (like mod switching), bootstrapping can be invoked an unlimited number of times, allowing FHE schemes to execute algorithms with *unbounded depth*. It can also be combined with mod switching, where mod switching is used as a first noise mitigation technique until no more primes can be removed, and then bootstrapping can be invoked. Bootstrapping will regenerate ciphertext primes, so modulus switching can be used in between subsequent bootstraps.

All LHE schemes can become FHE schemes with the inclusion of bootstrapping. Unfortunately, this ability comes with a steep cost: bootstrapping is the major bottleneck in all FHE schemes. For BGV, CKKS, and BFV, this procedure can take dozens of seconds to several minutes (depending on parameters). However, while newer schemes have improved bootstrapping latency, these schemes maintain the highest throughput per bootstrap due to batching capabilities.

The first efficient bootstrapping scheme is the FHEW cryptosystem [11], which introduced a much faster bootstrapping mechanism that takes less than a second to evaluate on a CPU. This scheme encrypts single bits of plaintext as individual ciphertexts and allows users to evaluate homomorphic Boolean gate operations. Contrary to prior schemes, bootstrapping plays a key role in evaluating the logic gates, so each gate operation requires a bootstrap. Further, this scheme was improved by TFHE [3], which accelerated bootstrapping to approximately ten milliseconds. FHEW is not configurable for users and TFHE only supports two monolithic parameter sets (one for 80 bits of security and the other for 128 bits) and therefore is trivial for developers to configure. As such, our methodologies presented in this article do not directly apply to these schemes in their current implementations.

III. LHE Parameterization

Implementations of BGV, BFV, and CKKS typically grant users a very high degree of freedom when it comes to customizing a wide variety of encryption parameters. Most importantly, the user can select the size of the prime chain (and hence the ciphertext modulus) and the degree of the irreducible cyclotomic polynomial (which serves to constrain the number of coefficients in ciphertext polynomials).

These two parameters must be balanced to achieve a suitable level of security. On one hand, having a longer prime chain allows one to increase the number of possible modulus switches, allowing for a greater number of leveled operations. However, the size of the prime chain has a negative impact on security and must be minimized to achieve just enough levels to evaluate any given application. On the other hand, increasing the polynomial degree has a positive impact on security. This parameter also serves to govern the speed of homomorphic operations; operations involving ciphertexts with larger polynomials will be slower.

In our experience, the most straightforward methodology for finding the optimal parameters for a given application involves first focusing on tuning the ciphertext modulus size and then altering the polynomial degree to achieve a desired security level and number of slots. The first step involves analyzing the application and identifying the *multiplicative depth*, i.e., the maximum number of multiplications that happen on a single ciphertext. For complex applications, this can be non-trivial, in which case a rough approximation can be used instead. Once this is determined, a general rule of thumb is that you should add d primes to the chain, where d is the multiplicative depth. The size of the primes varies per implementation, but usually the primes in the chain are chosen to be approximately the same size. However, there are some exceptions: in the case of the CKKS cryptosystem, for example, it is most common for the first and last prime to be larger than the intermediate primes. Consequently, since the composite number of the prime chain is the modulus of the polynomial coefficients of ciphertext polynomials, the larger the prime chain, the more memory is required to hold ciphertext data.

Once the bit size of the initial ciphertext modulus is decided, one can then run multiple iterations of the homomorphic program to fine-tune this value. It is recommended to use a small polynomial degree for these tests (i.e., up to degree 4096) which will yield low security in practice but can allow one to rapidly verify correctness. If the final decrypted answer matches the expected value for given input pairs across multiple runs, the ciphertext modulus is sufficiently large. Lastly, the modulus should be decreased until the answer becomes non-deterministic (meaning that the result is too noisy) and the last size that resulted in correct evaluation should be selected as the final modulus size.

IV. FHE Parameterization

Choosing optimal parameters for HE implementations with bootstrapping is largely orthogonal to the LHE case. Instead of choosing parameter sets that can evaluate circuits up to the required depth, one must choose parameters that minimize the bootstrapping cost. This is highly dependent on the implementation and underlying scheme. For instance, HElib incorporates two bootstrapping variants for BGV: a *full* bootstrap and a *thin* bootstrap. Thin bootstrapping requires sparsely packed ciphertext slots, but can achieve higher throughput (and an amortized cost per slot as low as 1 millisecond [12]). In this case, the slots are sparsely packed if each encodes an integer in $\mathbb{Z}p^r$. Conversely, CKKS bootstrapping restricts the precision of the underlying plaintext floating point numbers. With state-of-the-art methods, 40 bits is the maximum precision achievable in a bootstrapped CKKS context [13]. In terms of performance, the CKKS bootstrapping time scales linearly with the cyclotomic order and the number of slots.

Further, identifying the correct locations in the algorithm to insert a bootstrapping procedure requires close monitoring of the noise levels in the ciphertexts. Luckily, all libraries that support bootstrapping also incorporate this functionality. This is also necessary because the bootstrapping procedure will introduce additional noise before it can reduce the ciphertext noise levels. The noise cost of the bootstrap is determined solely by the parameter choices as well. We have found that the best way to minimize this cost is to use parameters with smaller plaintext moduli. Indeed, smaller moduli are necessary for efficient evaluation in BGV/BFV, as the noise cost per multiplication increases with higher plaintext moduli and the decryption circuit becomes deeper, restoring fewer overall levels per bootstrap. We discuss this further in the following section when evaluating bootstrap speeds for varying plaintext modulus sizes.

TABLE I

LHE PARAMETERS: THE L COLUMN CORRESPONDS TO THE NUMBER OF LEVELS ACHIEVABLE WITH THE CORRESPONDING PARAMETER SET. THE TOP HALF OF THE TABLE SHOWS OPTIMIZING FOR THROUGHPUT AND THE BOTTOM ARE PARAMETER SETS OPTIMIZED FOR LATENCY. "COST" IS THE AMORTIZED MULTIPLICATION COST FOR EACH SLOT.

L	m	N	$\log_2(q))$	Slots	λ	Mult Time (ms)	Cost (μs)
2	10923	6600	60	3300	164	44	13.3
9	23377	23040	360	3840	134	224	58.3
17	47333	39600	660	6600	134	664	100.6
30	65281	64512	1080	10752	128	1458	135.6
2	5461	5292	60	126	128	27	214.3
9	23377	23040	360	3840	134	224	58.3
17	40951	39600	660	30	134	614	20466.7
30	65281	64512	1080	10752	128	1458	135.6

V. EXPERIMENTAL EVALUATION

We ran experiments with the HElib library's BGV implementation in both LHE and FHE modes as a case study to demonstrate our methodology. To simplify the discussion, we implemented a small program that computes iterative multiplications between two integer ciphertexts and stores the product in one of the two ciphertexts. In this way, each multiplication iteration reflects one multiplicative level. All experiments were conducted single-threaded on a laptop with i7-8650U CPU (1.9 GHz) using HElib v2.2.1 and all parameters used correspond to at least 128 bits of security in both LHE and FHE contexts. Timing measurements were taken using the `high_ resolution_clock` class provided in the C++ standard library.

For LHE experiments, we aim to do multiplications over ciphertexts encrypting 16-bit integers. To enable this, we set our plaintext modulus p (which must be prime) to 65537; note that the underlying plaintext in a ciphertext can not exceed p, and will wrap around identically to standard modular arithmetic (i.e., $Enc(1000) * Enc(70) = Enc(4463)$). We consider parameter sets with two different (and sometimes conflicting) optimization goals: slot maximization and smallest polynomial degree. The former corresponds to maximizing throughput, while the latter corresponds to minimizing latency. In some cases, as shown in Table I, for a target number of levels, a single parameter set can satisfy both requirements. The procedure used to derive the experiments included compiling and running `HElib/misc/params.cpp` with our given value of p to generate candidate cyclotomic degrees. Next, we developed a Python module that takes the candidate m values and computes the number of slots available for each case with our given

TABLE II

FHE PARAMETERS: ALL PARAMETER SETS INCORPORATE $m = 55831$ (CORRESPONDING TO $N = 54000$), AND INITIAL CIPHERTEXT MODULUS SIZE OF $\log_2 q = 780$. ADDITIONALLY, THIS PARAMETER SET YIELDS 129 BITS OF SECURITY AND 2160 SLOTS.

p	Bootstrap Time (s)	Gained Levels	Amortized Cost (ms)
2^1	58.5	29	0.9
2^4	73.4	20	1.7
2^8	119.6	13	4.3
2^{12}	125.0	9	6.4
2^{16}	138.2	2	32.0

p. This is simply computed as the ring dimension N divided by the multiplicative order of $p \mod m$. When optimizing for latency, we chose the smallest value of m that yielded a sufficient ring dimension N. Conversely, when optimizing for slots we chose a value of m that yielded the highest number of slots for a ring dimension N that satisfies the security requirement for the necessary q. Lastly, we tuned q to achieve approximately 128 bits of security when possible with the given value of N, and identified four distributed parameter sets that can achieve a varying number of levels up to 30. Overall, we observe that being able to achieve a higher number of multiplications necessarily requires a larger ciphertext modulus. Consequently, the polynomial degree m must be increased to yield a higher ring dimension N (defined as $\phi(m)$), to maintain a high level of security. The increased polynomial degrees and coefficient sizes result in increasingly slower multiplication speeds. We report the cost of the first multiplication in the program for each parameter set, but we note that subsequent multiplications become slightly faster as the ciphertext modulus becomes smaller over time due to modulus switching, which is invoked after each multiplication.

In the FHE case, we performed slight modifications to the multiplication loop to enable infinite multiplications; in every iteration we check the remaining noise capacity in the product ciphertext and when it drops below 100 bits of capacity, we perform bootstrapping to refresh the noise. Further, we use the same core parameter set to demonstrate performance with respect to different plaintext spaces. The plaintext modulus is set to $p = 2$ in all cases, but we use a technique provided by HElib called *Hensel lifting* to extend the plaintext space to higher powers of 2. This way, we are also able to do multiplications over higher bit numbers, as opposed to just modulo 2 (which is essentially equivalent to a logical `AND` gate). In order to achieve appreciable gains from bootstrapping, we opt to use a large initial ciphertext modulus (780 bits) which necessitates using a large cyclotomic degree (55831) that yields a high ring dimension of $N = \phi(55831) = 54000$ to achieve 129 bits of security. We found that using a smaller q actually had a detrimental effect when bootstrapping was invoked and resulted in a net loss in terms of noise budget. Table II shows both the thin bootstrapping latency for different plaintext sizes and how many levels are restored after bootstrapping. Essentially, this indicates the number of multiplications that can occur in between bootstraps. The lowest amortized cost we achieve at 129 bits of security is roughly 0.9 milliseconds for GF(2), where the amortized cost is defined as the bootstrapping time divided by the number of slots and further divided by the number of restored levels. This follows the definition introduced by Halevi and Shoup in their work outlining bootstrapping techniques in HElib [12].

VI. Related Work

Prior works have tackled the issue of HE parameterization for specific use-cases. The HE-PTune module of Cheetah [14], a framework for neural network inference using homomorphic encryption, automatically chooses HE parameters to optimize performance for neural network layers. HE-PTune uses cost models derived for fully-connected and convolutional layers to compute the parameter set that yields the smallest remaining noise budget at the end of each layer. However, Cheetah assumes only an LHE context and is built specifically for parameterization for neural network layers, not arbitrary applications.

Another work called CinguParam [15] focuses on generating a database of parameter sets for BFV and introduces noise models for multiple key switching mechanisms. While CinguParam can provide parameter sets for many applications developed with BFV, it only considers relatively "shallow" algorithms with a multiplicative depth less than 21. Contrary to this, we provide guidelines for selecting parameter sets for both LHE and FHE, making our insights valuable for any application regardless of multiplicative depth.

Lastly, the Academic Consortium to Advance Secure Computation has introduced general HE parameter sets achieving 128, 192, and 256 bits of security. In practice, these parameter sets serve as good starting points for exploring parameters for a given application, but do not necessarily include the most optimal for a given algorithm. Further, they provide no guidance for choosing which of the parameters are most appropriate.

VII. Conclusion

Usability is one of the largest hurdles that HE needs to overcome to see widespread adoption. In this article, we investigate this problem by introducing guidelines, insights, and heuristics for choosing efficient parameters for state-of-the-art homomorphic encryption libraries. In addition, we have conducted experiments using our parameterization methodologies with the BGV cryptosystem implemented in the

HElib library and identified several viable secure parameter sets for both LHE and FHE. By following the methodology presented here, developers can create optimal homomorphic implementations that take full advantage of this powerful, privacy-preserving technology.

References

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *ACM Symposium on Theory of Computing*, 2009, pp. 169–178.

[2] A. Greenberg, "IBM's Blindfolded Calculator." [Online]. Available: https://www.forbes.com/forbes/2009/0713/breakthroughs-privacy-super-secret-encryption.html?sh=5a71e2a37124

[3] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: fast fully homomorphic encryption over the torus," *Journal of Cryptology*, vol. 33, no. 1, pp. 34–91, 2020.

[4] L. Folkerts, C. Gouert, and N. G. Tsoutsos, "REDsec: Running Encrypted Discretized Neural Networks in Seconds," Cryptology ePrint Archive, Report 2021/1100, 2021.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[6] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[7] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1999, pp. 223–238.

[8] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009.

[9] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *Cryptology ePrint Archive*, 2012.

[10] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2017, pp. 409–437.

[11] L. Ducas and D. Micciancio, "FHEW: bootstrapping homomorphic encryption in less than a second," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2015, pp. 617–640.

[12] S. Halevi and V. Shoup, "Bootstrapping for HElib," *Journal of Cryptology*, vol. 34, no. 1, pp. 1–44, 2021.

[13] J.-W. Lee, E. Lee, Y. Lee, Y.-S. Kim, and J.-S. No, "High-precision bootstrapping of RNS-CKKS homomorphic encryption using optimal minimax polynomial approximation and inverse sine function," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2021, pp. 618–647.

[14] B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 26–39.

[15] V. Herbert, "Automatize parameter tuning in Ring-Learning-With-Errors-based leveled homomorphic cryptosystem implementations," *Cryptology ePrint Archive*, 2019.