# Implementing and Benchmarking a LWE-based Fully Homomorphic Encryption Scheme

Meghan L. Clark
*University of Michigan*

Alex L. James
*University of Michigan*

Travis B. Martin
*University of Michigan*

## Abstract

Fully homomorphic encryption (FHE) provides a way for third parties to compute arbitrary functions on encrypted data. This has the potential to revolutionize cloud computing services. Unfortunately, since their emergence in 2009, FHE schemes have become notorious for incurring enormous costs in time and space. Over the last four years optimizations have been proposed with impressive rapidity. However, these improvements are usually only asymptotically beneficial. The newness of the field and relative opacity of the literature has resulted in few implementations and evaluations of actual performance. To fill this gap, we implement a recent FHE scheme based on the Learning with Errors (LWE) hardness problem. We compare the performance of our system with an implementation of an earlier FHE scheme based on the Approximate GCD (AGC) hardness problem. We find that the LWE scheme performs much better than the AGCD scheme for anything larger than moderately sized security parameters. We release our system to the public to promote additional experimentation and to increase the accessibility of this new cryptographic construct.

## 1 Introduction

Fully homomorphic encryption (FHE) is a cryptographic scheme where any function can be computed on encrypted data, and the decrypted result will be the correct answer.

FHE has profound implications for cloud computing services. The security and privacy concerns associated with providing your data to a third party are eliminated, as they will never see your decrypted data, and neither will anyone who compromises their system. Some applications of this are with medical records - hard to get a hold of, run statistics on, but no longer. Satellites - no longer need to trust a third party to keep secrets. Private search queries. Private location-based-services. Secure voting.

Unfortunately, in their current state, FHE schemes incur too much overhead to be used in actual applications. Even in some extremely recent examples, such as evaluating AES-128 encryption where they increased performance by orders of magnitude, the process required over three days to complete and 256 GB of RAM[7].

Since the first FHE scheme was proposed in 2009, combating the performance costs has been the primary focus of the FHE research community. They propose many optimizations, but mostly asymptotic results. Since they are theory folks and optimizations are coming out so rapidly, there have been few implementations. However, this means that it is hard to gauge whether or not optimizations that are asymptotically faster represent actual performance improvements. Without this feedback, it's hard to tell which branch of FHE to pursue.

¡¡¡¡¡¡¡ HEAD Recently a scheme was proposed by Brakerski, Gentry, and Vaikuntanathan in [5] that involves a new branch of FHE based on LWE that is able to operate without bootstrapping. The authors state explicitly that while their solutions are asymptotically faster, they don't know whether or not they're actually faster in practice than earlier schemes: ======= Recently a scheme was proposed by Brakerski, Gentry, and Vaikuntanathan in [5] that involves a new branch of FHE based on Learning with Errors that is able to operate without bootstrapping. The authors state explicity that while their solutions are asymptotically faster, they don't know whether or not they're actually faster in practice than earlier schemes: ¿¿¿¿¿¿¿ 72719b25a301a88cdcf753b9b0262c788fa948e7

> "Performance-wise, this scheme trounces previous (bootstrapping-based) FHE schemes (at least asymptotically; the concrete performance remains to be seen)."

We answer this call by implementing it and comparing it to a bootstrapping AGCD-based scheme by Coron, Naccache, and Tibouchi. This comparison adds to the

sparse set of real-world FHE performance data points. We are also the first to release LWE FHE code to the public. This can be used for additional parameter exploration, cracking attempts, or a hands-on introduction to an FHE system.

## 2 Background

In this section we provide a brief history and an explanation of the mechanics of the two main FHE schemes. This will provide the background required for understanding the FHE implementations that we evaluate.

### 2.1 Circuits

The goal of FHE is to be able to perform arbitrary computations on ciphertext, such that the decrypted result is the same as the result of performing the same computations on the plaintext.

To be capable of arbitrary computation, the cryptosystem need only support two operations, which we will call addition and multiplication, such that

$$D_k(E_k(m_1) + E_k(m_2)) = m_1 + m_2$$

$$D_k(E_k(m_1) * E_k(m_2)) = m_1 * m_2$$

where $E$ and $D$ are the encryption and decryption functions.

It has been shown elsewhere that supporting these two operations provides Turing completeness. However, for an intuitive explanation, consider that your computer can run arbitrary programs, yet all of the logical circuits in the hardware can be theoretically implementing using just AND and XOR gates.

For this reason, in the FHE literature all functions to be computed on ciphertext are called "circuits" and are constructed as nested binary operations, such as $(E_k(1) + ((E_k(1) * E_k(0) + E_k(1))$.

Following suit, throughout the rest of this paper we assume that the operands $m_1$ and $m_2$ are single bits (either 0 or 1), and we assume that the addition and multiplication operations are equivalent to binary addition (XOR) and binary multiplication (AND).

### 2.2 Partial Homomorphism

Cryptographic schemes that support just one of the two necessary operations are called *partially homomorphic*. The possibility of FHE was first suggested after examining the partially homomorphic properties of RSA in 1978 [9], although an actual FHE scheme was not successfully devised until over thirty years later.

A cryptographic scheme that supports two operations but only for a limited number of successive operations is called *somewhat homomorphic*. A somewhat homomorphic encryption (SWHE) scheme was crucial to the development of the first FHE scheme and in most every subsequent scheme.

### 2.3 Lattices

The first working FHE scheme was proposed in 2009 by Craig Gentry [3, 4]. The system was based on a mathematical construct called lattices. In particular, the security of the scheme rested on the hardness of solving certain problems using lattices. However, this system was so complicated and difficult to understand that it immediately gave way to equivalent yet more intuitive integer-based schemes [10, 12] which we will describe in detail in the next section.

Though the FHE literature quickly shifted away from representing ciphertext and keys using lattices, it is worth noting that all current FHE schemes can still ultimately trace their security back to lattice-based hardness problems.

### 2.4 Approximate GCD Schemes

A year after Gentry described the first FHE scheme, he released a second scheme that translated his lattice-based system into an integer-based system whose security relied on the Approximate GCD hardness problem [12]. This system spawned a family of related AGCD schemes that tweak or optimize Gentry's original scheme.

Gentry described a three-step process for constructing a fully homomorphic scheme. First he began with a SWHE scheme. The SHWE scheme he describes stops working after only a few optimizations because a noise term in the ciphertexts grows and overwhelms the rest of the message. You could refresh the noise by decrypting and re-encrypting the message. However, this is undesirable for third party computing. However, since decryption is a mathematical function, you could decrypt *homomorphically* with an encryption of the private key. This would take a ciphertext and produce another ciphertext with reduced noise. However, the decryption circuit must take few enough operations that it introduces less noise than it takes out. So the second step is to "squash" the decryption circuit by transforming the problem from its current encrypted space into a smaller encrypted space with much less complexity. After this is done it is possible to perform the third step, homomorphically decrypting the ciphertext to refresh the noise, effectively *bootstrapping* the result into a much smaller noise range for more computation.

#### 2.4.1 Somewhat Homomorphic Encryption Scheme

First, we begin with the somewhat homomorphic design over the integers. Consider the encryption scheme $E_k(m) = m + 2e + pq$ where the private key $q \leftarrow G(1^\lambda)$, $p < q$ an odd integer and the noise term $e << p$. Then this scheme will encrypt a single bit message which can be recovered with the decryption scheme $D_k(c) = (c \bmod p) \bmod 2$.

One can see that addition satisfies the homomorphism property that we are after. Note that the noise term grows, albeit minimally:

$$E_k(m_1) + E_k(m_2) = m1 + m2 + 2(e_1 + e_2) + p(q_1 + q_2)$$

Similarly, multiplication also has this property, but with a caveat attached. By multiplying our ciphertexts we have also multiplied our error terms together, in very few multiplication steps, this error can grow out of control and quickly dwarf $p$. Once this happens, decryption fails as the number wraps around $\bmod p$

The security of this SWHE scheme reduces to the difficulty of the Approximate Greatest Common Divisor problem, a problem which states that given $n$ near divisors of $p$ it is still difficult to compute $p$ if $p$ is relatively large.

#### 2.4.2 Squashing the decryption circuit

So we are able to do many additions, but only a few multiplications due to exponential growth of the noise term. It is therefore necessary to "squash" the decryption circuit. Recall that the traditional way to represent the decryption circuit for this scheme was with

$$m = (m + 2e + pq \bmod p) \bmod 2$$

This modular arithmetic can also be expressed as $c - p\left\lfloor \frac{c}{p} \right\rfloor = m + 2e$. Given that we are working with bits, this is the same as $m = LSB(m + 2e) = LSB(c) - LSB(p\left\lfloor \frac{c}{p} \right\rfloor) = m$. Taking this to be our new decryption function, we can further reduce the number of operations it requires to homomorphically decrypt by giving the third party a representation of $\frac{1}{p}$.

This is done without revealing $p$ by taking advantage of the hardness of the Sparse Subset Sum Problem (SSSP). First construct a large vector $S$ which contains encrypted bits, some small subset of which sums to $\frac{1}{p}$. Next, construct a secret vector $v$ that serves as a bitmask that reveals which bits are summed to produce $\frac{1}{p}$. I.e., $v_i = E_p(0)$ if the $i^{th}$ element of $S$ is not used in the sparse subset and $v_i = E_p(1)$ if it is. Notice that $S * v = E_p(\frac{1}{p})$. In this way, the third party can use $\frac{1}{p}$ without knowing $p$.

#### 2.4.3 Bootstrapping

¡¡¡¡¡¡¡ HEAD Using a homomorphic implementation of the squashed decryption circuit $m = LSB(m + 2e) = LSB(c) - LSB(p\left\lfloor \frac{c}{p} \right\rfloor) = m$ and $\frac{1}{p}$, a third party can now periodically refresh the noise of intermediate results, or *bootstrap*, allowing indefinite computation without ever seeing plaintext or the secret key. ======= With this information the third party is able to evaluate the homomorphic decryption circuit without having to compute either $p$ or $\frac{1}{p}$, reducing the number of computations, and so the depth of the circuit. ¿¿¿¿¿¿¿ 72719b25a301a88cdcf753b9b0262c788fa948e7

#### 2.4.4 Problems

Gentry's bootstrapping theorem was key to creating a FHE scheme from a SWHE scheme. Unfortunately, there are many downsides to bootstrapping. The most costly downside is that bootstrapping is a very expensive operation in general. It requires a large amount of time to run[6] and also can lead to large ciphertexts. Even with this in mind, many of the early fully homomorphic encryption schemes bootstrapped after every operation due to the large amount of noise introduced by their multiplication operations. There have been several recent advancements in implementations using bootstrapping[11, 10, 6, 2] but the research community has also investigated whether it is possible to remove the necessity for bootstrapping entirely.

One family of homomorphic schemes departed from the AGCD-based schemes like Gentry's and based their security on the hardness of Learning with Errors (LWE). This family of schemes recently produced a bootstrapless implementation, which we describe next.

### 2.5 Learning With Errors Schemes

Using a form similar to Gentry's AGCD-based scheme, the Learning With Errors (LWE) schemes[5] viewed their ciphertext as randomly chosen vectors dotted and added with a secret key.

$$E_s(m) = (a, < a, s > +2e + m)$$

$$D_c(x) = (< a, s > +2e + m) - < a, x >$$

The Learning With Errors problem is a very powerful problem to base cryptographic constructions on. It is rather famous for being as hard as worst-case lattice problems, allowing all constructions based solely on it to be secure under the assumption that worst-case lattice problems are hard.

In this scheme, addition of two ciphertexts is done in the same way that addition would naively be done. However, when looking at the multiplication it is helpful to

look at the invariant linear function $\phi(x) \equiv b - < a, x >$ and multiply these instead. When this is done it takes the key from dimension $n$ to dimension $n^2$, so to maintain manageable key sizes for even small circuit depth a relinearization step is necessary. For this to happen, it is necessary for the client to post encryptions of pairwise multiples of their secret key's entries. Luckily, the rings we are dealing with are commutative and so it is only necessary to post $E(s[i]s[j])$ for $0 \leq i \leq j \leq n$ where $s[0] \doteq 1$. This allows transformation from a keyspace in $s$ of degree $n^2$ to a second keyspace for a secret key $t$ of size $n$. If we assume circular security, then we can simply publish this transformation rather than a large chain of transformations and $s$ will then re-encrypt to itself.

However, this does not negate the impact of noise, as the chain progresses, the noise will grow. To handle the noise it is helpful to observe the ratio between the noise $B$ and the modulus $q$. When $\frac{B}{q} \approx 1$ the noise overwhelms the message. We would prefer to have a small noise ratio. This is done with a well-known property of modulus, if $ab \equiv ac \bmod ad$ then $b \equiv c \bmod d$. As in much of this work, we do not need exact precision as we are already dealing with an error term. Instead we can approximate this effect even if the terms do not divide as nicely. By doing this, we reduce the absolute value of the noise from $B^2$ after a multiplication back down to $B$. This allows us to evaluate $L$ depth circuits with modulus $q \geq B^L + 1$ rather than having to require a modulus $q \geq B^{2^L}$. This method is called *modulus reduction*.

To sufficiently lower the complexity of the decryption circuit it is also necessary to decrease the dimension of the ciphertext which we are dealing with. This is done with *dimension reduction*. One should note from the relinearization stage that it is not necessary to have $s$ and $t$ be of the same dimension. Using this fact, it is possible to convert the ciphertext from using a private key $s$ of dimension $n$ to a private key $t$ of dimension $k$. After these steps have been performed our ciphertext will have reduced noise and reduced complexity, which allows for bootstrapping as an optimization.

The last statement might seem out of place to the reader, as bootstrapping was just described as being very inefficient. However, it can be an optimization if the depth of the circuit is large enough that evaluation on the high-dimension and high-modulus components becomes restrictive. In this case, evaluation on a smaller circuit with bootstrapping to refresh the noise would in fact be an optimization rather than a bottleneck.

## 3 Related Work

There are a few implementations of FHE schemes that have been used in papers over the past four years. However, many of these implementations have been rough prototypes, provided little performance data, or have not been released publicly.

In [8], the authors provide a rough proof-of-concept of a FHE scheme based on the Ring-LWE hardness problem without any optimizations.

In [6], bootstrapping an AGCD-based scheme took between 30 seconds for toy examples and 30 minutes for larger examples. In [7], a Ring-LWE-based homomorphic implementation of the AES circuit either took 150 hours to generate the key schedule or a time-out occurred before even a single iteration completed. These papers only offer rough benchmarks for their specific scheme. We intend this project to provide a point of reference for this scheme with hopes that future improvements can be easily added by either the authors or by third parties.

## 4 Methodology

We began this project with a thorough investigation of FHE schemes in general, theoretical attacks on insecure schemes, and hardness assumptions and security parameters that current schemes are being based on. With this detailed background we moved toward an implementation stage by taking an existing FHE implementation written in Python SAGE and evaluating its performance. For a fair comparison to this method we then implemented our own version of the standard LWE FHE encryption scheme, also in Python SAGE.

To establish test cases, it was necessary to observe the size of the security parameter in the AGCD implementation and establish an equivalent security parameter to run our LWE implementation with. The other parameters of our scheme, such as the dimension of the original secret key, the size of the modulus and the range of the error term are all determined by our security key in relations established by the implemented paper[1]. We then evaluated several test cases to benchmark our implementation against theirs using these parameters. These test cases are described in Section 6.

To emphasize the validity of our comparison, we note that both implementations were straightforward extensions of mathematically specified schemes from their respective papers. Further, each scheme was evaluated for the same set of security parameters, with additional parameters chosen to give the desired level of security. Additional parameters were chosen according to the provably secure ranges provided in [1, 2]. Our tests were run in similar environments on the same computer. Due to these similarities, we reduce the possibility for performance differences to either differences in fundamental efficiency or differences in programming efficiency. These are discussed in Section 6.

## 5 Implementation

The code works in several stages. A user-level interface has been designed to support modifying the parameters used in the encryption process and parsing expression inputs for evaluation. The interface provides a simple and convenient way for users to verify that arbitrary functions can be applied to arbitrary inputs. In practice, function specification would be separate from data encryption so that multiple functions could be applied to the data at arbitrary points in time.

After the parser establishes the depth of the circuit, a list of public information is generated which transform the ciphertext between the necessary keys throughout the algorithm's multiple relinearization and dimension-modulus reduction operations.

All random variables are generated uniformly within an appropriate range, for instance, the error term is not allowed to be large relative to the modulus $q$ and so it was generated within $\log(q)$. This reinforces the hardness of the LWE problem by using the same parameter space that is used in the classical problem.

Once the public keys and supporting information have been generated, the circuit is evaluated. Throughout the process encryption and decryption are handled in the invariant form as elements of $\mathbb{Z}_q^n$ where multiplication is handled by a tensor product to $\mathbb{Z}_q^{n\times n}$ followed by relinearization back to elements of $\mathbb{Z}_q^n$ with degree less than $n$ and a modulus reduction to $\mathbb{Z}_p^k$ to reduce the noise incurred by a multiplication. This process is repeated until the final value has been computed. Modulus-dimension reduction and relinearization are both explained in Section 2.5, and optimizations proposed by Brakerski et al. [1]. Once the final ciphertext is computed, our user-interface decrypts the result to verify that the circuit has been computed correctly.

## 6 Results

Our implementation test consisted of two batteries of tests for varying security parameters. In the first, designed to compare our implementation to the implementation by Coron et al [2], we analyze the running time for adding and multiplying one encrypted bit. All times include time to encrypt, perform the respective operation and decrypt. Their multiplication time also includes time for bootstrapping. Our multiplication time also includes time for relinearization and modulus-dimension reduction. We don't include these operations in the addition operation timings because they aren't required due to minimal noise increase. All parameters are chosen to give the level of security required by the security parameter, according to guidelines specified by the original authors. Timings can be found in Figure 1. All tests were run on a Intel i7 2.67 GHz Quad Core processor with 24 GB of RAM.

As our figure shows, their performance is much greater for small security parameters. This is mainly due to the large difference in Sage programming experience between Coron et al. and the authors of this report. We believe there are many more efficient ways to express the polynomial operations we used, but we did not have time to optimize these operations. However, we see that, asymptotically, our implementation outperforms theirs. For large security parameters, the inherent efficiency advantage of learning with errors over approximate gcd becomes apparent. Additionally, we see that, in the limit of large security parameter, their addition and multiplication operations take about the same time. This is because the time of encrypting extremely large ciphertexts swamps the time required for bootstrapping. In the learning with errors based scheme, contrarily, multiplication remains approximately twice as slow as addition, even in the limit of large ciphertexts.

Our second set of tests attempts to analyze the efficiency of our scheme for deeper circuits. We first show that our scheme scales well - executing functions with several multiplications and additions takes only a small amount more time than performing one multiplication alone. Secondly, we find that deeper circuits, requiring more modulus-dimension reduction, are not necessarily slower than circuits with more operations. Results for the second set of tests can be found in Figure 2.

## 7 Discussion

The results that we encountered were not exactly what we were hoping for, as we were hoping to have shown that the LWE scheme not only outperforms asymtotically but also definitively in much smaller cases. This however does demonstrate one of the hardships of creating a fully homomorphic cryptographic system; it is necessary to understand the language which you are working in fully to take advantage of all optimizations available and to avoid all pitfalls.

We feel it necissary to reiterate that even with these pitfalls, our code does scale better with increased security parameters (fortunately this is where cryptography is done). This is a very good indication that LWE is indeed an asymptotically better solution than AGCD and we feel that further increasing the performance of our code will undoubtedly force the break even point much closer to the toy example given.

## 8 Future Work

There are several goals for the future. For this specific code two main goals are to refine it into a more module and adaptable structure for easy modification by a third party for review and experimentation and to optimize it by relying more heavily on Sage structures and experimenting with the fastest methods to do specific tasks. There are also a few more optimizations such as determining minimalistic parameters based on the depth of the circuit.

Further work can be done to implement and publish other forms of FHE encryption with their associated optimizations, forms such as the AGCD described above and the NTRU-like scheme.

## 9 Conclusion

Despite the alacritous pace at which theoretical work is being done to improve the performance of fully homomorphic encryption, no one is doing sanity checks by implementing these schemes. This makes it difficult to know what actual progress is being made and which directions are most promising. To address this deficit, we implemented a LWE-based FHE scheme in Python and Sage and compared it to an existing implementation of an AGCD-based FHE scheme, also written in Python and Sage. We found that the LWE implementation performed asymtotically better than the AGCD implementation by a very quickly increasing margin. In addition to expanding the corpus of FHE performance benchmarks, we also publicly release our code. We hope that this work helps provide direction to future FHE research and increases the accessibility of this new and exciting area of cryptography.

## 10 Availability

We feel that releasing our implementation to the public for examination and experimentation is one of the major contributions of this work. Our code and directions for running it can be found at:

`https://github.com/tbmbob/bootstraplessfhe`

If you have questions, we will do our best to answer them.

## References


[1] BRAKERSKI, Z., AND VAIKUNTANATHAN, V. Efficient fully homomorphic encryption from (standard) lwe. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on* (2011), IEEE, pp. 97–106.

[2] CORON, J.-S., NACCACHE, D., AND TIBOUCHI, M. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *Advances in Cryptology–EUROCRYPT 2012*. Springer, 2012, pp. 446–464.

[3] GENTRY, C. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[4] GENTRY, C. Fully homomorphic encryption using ideal lattices. In *STOC '09: Proceedings of the 41st Annual ACM Symposium on Theory of Computing* (2009), pp. 169–178.

[5] GENTRY, C. Fully homomorphic encryption without bootstrapping. *Security 111*, 111 (2011), 1–12.

[6] GENTRY, C., AND HALEVI, S. Implementing gentrys fully-homomorphic encryption scheme. In *Advances in Cryptology–EUROCRYPT 2011*. Springer, 2011, pp. 129–148.

[7] GENTRY, C., HALEVI, S., AND SMART, N. P. Homomorphic evaluation of the aes circuit. In *Advances in Cryptology–CRYPTO 2012*. Springer, 2012, pp. 850–867.

[8] NAEHRIG, M., LAUTER, K., AND VAIKUNTANATHAN, V. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop* (2011), ACM, pp. 113–124.

[9] RIVEST, R. L., ADLEMAN, L., AND DERTOUZOS, M. L. On data banks and privacy homomorphisms. *Foundations of secure computation 32*, 4 (1978), 169–178.

[10] SMART, N. P., AND VERCAUTEREN, F. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography–PKC 2010*. Springer, 2010, pp. 420–443.

[11] STEHLÉ, D., AND STEINFELD, R. Faster fully homomorphic encryption. In *Advances in Cryptology-ASIACRYPT 2010*. Springer, 2010, pp. 377–394.

[12] VAN DIJK, M., GENTRY, C., HALEVI, S., AND VAIKUNTANATHAN, V. Fully homomorphic encryption over the integers. In *Advances in Cryptology–EUROCRYPT 2010*. Springer, 2010, pp. 24–43.

| Example size | Security Parameter | Their add (s) | Their mult (s) | Our add (s) | Our mult (s) |
|---|---|---|---|---|---|
| Toy | 42 | .048 | .560 | 18.0 | 35.9 |
| Small | 52 | 3.38 | 7.37 | 48.9 | 98.6 |
| Medium | 62 | 71 | 106 | 110.3 | 225.1 |
| Large | 72 | 1460 | 1860 | 249 | 524.3 |

Figure 1: Timing: Coron et al. implementation vs. our LWE-based implementation, for varying security parameters

| Example size | Security Parameter | 11 operations, depth 1 (s) | 7 operations, depth 2 (s) |
|---|---|---|---|
| Toy | 42 | 42.5 | 39.4 |
| Small | 52 | 109.3 | 109.5 |
| Medium | 62 | 251.0 | 251.1 |

Figure 2: Timing: our LWE-based implementation for varying numbers of operations, depth