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 [BETTER/WORSE] than the AGCD scheme. 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 encrpytion where they increased performance by orders of magnitude, the process required over three days to complete and 256 GB of RAM[6].

Since the first FHE scheme was proposed in 2009, combatting 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.

Recently a scheme was proposed by Brakerski, Gentry, and Vaikuntanathan in [4] that involves a new branch of FHE based on LWE 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:

> "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 paramter 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 possiblity of FHE was first suggested after examining the partially homomorphic properties of RSA in 1978 [8], although an actual FHE scheme was not successfullly 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.

2.3 Lattices

The first working FHE scheme was proposed in 2009 by Craig Gentry [2, 3]. 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 [9, 11] 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 [11]. 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. However, as his thesis points out, when your system can evaluate its own decryption circuit, you become powerful enough to indefinitely compute. Which is what we want, however, as the scheme is only somewhat homomorphic it is the case that its decryption circuit is too complicated and requires too much nose introduction through multiplications. 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 homomorphically decrypt the ciphertext to refresh the noise, effectively bootstraping 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 $qleftarrowG(1^\lambda)$, $p < q$ 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 both addition satisfies the homomorphism property that we are after; doing so with very min-

imal noise increase.

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

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 $\bmod p$

It is important to note that the security of this scheme is directly replated 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.

Start with SWHE scheme (describe. You can use the real public key, the real encryption with the zeros from the public key if you want, the decryption function, etc. Get mathy!). Mention AGCD hardness assumption.

2.4.2 Squashing the decryption circuit

So we are able to do many additions, but only a few multiplications due to exponential growth. It is therefore necissary to "squash" the decryption circuit. The standard way to approach squashing this particular scheme is to give hints to the private key in the form of $E_p(p)$ and a Sparse Subset Sum problem involving $\frac{1}{p}$. With this additional information, homomorphic decryption becomes much more accessible and has a much smaller circuit.

2.4.3 Bootstrapping

What exactly does the decryption circuit look like? Remember that the traditional way to represent the decryption circuit for this scheme was with

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

However, we can not have the third party directly computing $\bmod p$ as that would entirely reveal our secret key p. Instead this modular arithmetic can be phrased as $c - p\lfloor(\rfloor \frac{c}{p}) = m + 2e$. Finally, after noticing that we can see then, $m = LSB(m+2e) = LSB(c) - LSB(p\left\lfloor \frac{c}{p} \right\rfloor) = m$. If it is possible to provide the necissary information for this method without revealing information about the private key, then it is possible to homomorphically decrypt.

As mentioned above, this is done though the sparse subset problem. Encrypt values into a large encrypted vector S which has a very sparse subset summing to $\frac{1}{p}$. Next, construct a secret vector v such that $v_i = E_p(0)$ 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 $S.v = E_p(\frac{1}{p})$, however the third party does not know which subset of either vector was used in the computation.

With this information the third party is able to evaluate the homomorphic decryption circuit without having to compute $\frac{1}{p}$, reducing the number of computations, and so the depth of the circuit.

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[5] and also can lead to large ciphertexts[**?**]. Even with this in mind, many of the early families of fully homomorphic encryption schemes bootstrapped after every operation due to the large amount of noise introducted by their multiplication operation[**?**]. There have been several recent advancements in implementations using bootstrapping[10, **?**, 5, 1] but the research community has also focused intently on removing the necessity for bootstrapping entirely such as in Learning with Errors, the scheme that was implemented by this paper.

2.5 Learning With Errors Schemes

In addition to AGCD-based schemes, another family of schemes arose which made intuitive sense. Using a similar form to the AGCD-based scheme, the Learning with Errors Schemes[4] schemes viewed their ciphertext as randomly chosen vectors dotted and added with a secret key.

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

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

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 invarient 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 and so to maintain managable key sizes for even small circuit depth a relinearization step is necissary. For this to happen, it is necissary for the client to post encryptions of pairwise multiples of their secret key's entries. Luckily, the rings we are dealing with are communitive and so it is only necissary 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 and so we prefer to have a small noise ratio. This is done with a well known property of modulus, if $ab \equiv ac \bmod ad$ then $b \equiv c \bmod d$. Just like 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 lower the complexity of the decryption circuit a sufficient amount it is also necissary to lower 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 isn't necissary 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.

Out implementation of the LWE scheme is as described above, in it we implement multiple optimizations to allow for quick computation of the circuit such as the modulus scaling property.

3 Related Work

There are a few implementations of FHE schemes that have been used in papers over the past four years. Specifically, implementations such as the evaluation of the 128-AES circuit[6] where the authors aimed to show that their scheme was powerful enough to handle the AES circuit, Implementing Gentry[5], the "Practical" paper[7], and the Coron and Scarab projects. All of these have implemented various FHE encryption schemes and have shown optimizations over previous versions while still having their drawbacks. For instance, bootstrapping taking between 30 seconds for small examples and 30 minutes for larger examples[5] and generation of the key scheduling taking 150 hours or evaluation timing out before even a single iteration is complete[6]. These implementations attempt to improve upon previous works and offer a quick benchmark for the possible implementation of the scheme before moving onto the next. We intend this project to provide a static point of reference for this scheme with hopes that future improvements can be easily added by either the authors or by third parties.

This section is for highlighting what makes our work different from previous work.

4 Methodology

This is about the experimental design. Start off with a super high-level description of our approach. A.k.a, given an existing Python SAGE implementation for AGCD, write our own LWE implementation also in Python SAGE, and then benchmark each on a variety of parameters described here.

IMPORTANT: This is where we make the case that our benchmarking comparisons are fair. This whole section is not just describing but also *justifying* our experimental design choices. Namely, the optimizations we chose, the parameters we chose, etc., and why those choices make it fair.

This is the most important section of the paper. This is where the reader can tell whether we did good science or bad science.

5 Implementation

The nitty-gritty about the implementation. This can just be a straight-up description about how our code works.

If you want to include code snippets, here's some example LaTeXfor that:

```
int main() {
    int result = 1
    if (result != 2) {
        printf("You get here every time.\n");
    }
}
```

6 Results

Here's a typical figure reference. The figure is centered at the top of the column. It's scaled. It's explicitly placed. You'll have to tweak the numbers to get what you want.

This text came after the figure, so we'll casually refer to Figure 1 as we go on our merry way.

7 Discussion

Can be short. Talk about the main results, particularly the impact. Were our results reasonable or surprising?

Figure 1: Description of what information is being displayed. Results/trends of note. What those results/trends mean (impact).

Do our results mean that we should advocate for LWE or AGCD? Are they too close to call? Or are our results too specific to the implementations and optimizations used? When might we get different results?

8 Future Work

The future work for this specific code is to refine it into a more module and adaptable structure for easy modification by a third party for review and experiementation. Additionally, a few more optimizations and code timing analysis will be necissary to ensure that the code is running as quickly as it can at the hardware level. 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.

Can be short. Stuff not only that you'd want to do on this project in the future, but what any FHE researcher might try next after having read this.

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 [BETTER? To WHAT DEGREE?] than the AGCD implementation. 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 very strongly 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] 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.

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

[3] 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.

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

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

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.