Algorithms for Evaluating Nonlinear Functions Under Homomorphic Encryption

Trevor Henderson
Applied Cryptography Group
Orange Labs, Caen, France

August 30, 2016

Abstract

This paper investigates methods of computing nonlinear functions efficiently under the constraints of common fully homomorphic encryption schemes — access to only addition and multiplication modulo t. I show that $\sqrt{t}$ multiplications are necessary and $3\sqrt{t}$ multiplications are sufficient to evaluate a general single variable function. I then show that many multivariate functions, such as division and boolean comparisons can also be evaluated using $O(\sqrt{t})$ multiplications. The multiplicative depth of both algorithms is $O(\log t)$. Despite meeting asymptotic lower bounds, it is disappointing to learn that any reasonably complex homomorphic function is necessarily weakly polynomial. This suggests that more flexible encryption schemes must be pursued if homomorphic encryption is to become practical.

Experimental results utilizing an implementation of the somewhat homomorphic scheme, YASHE, demonstrate secure, homomorphic evaluations of arbitrary 8-bit single variable functions in less than 4 milliseconds amortized on a 2012 MacBook Pro. Test scenarios that could be useful in cloud computing, such as converting an image from RGB to YCbCr color formats and applying color filters, are computed in minutes. Both of these results show significant improvement over the state of the art.

1 Introduction

Put simply, homomorphic encryption allows one to encrypt messages and then perform math on these encrypted messages without revealing the contents [12]. Researchers speculate that if the field becomes practically feasible it could revolutionize the security of cloud computing [18]. For example, imagine a search engine that allowed you to encrypt a query with homomorphic encryption before sending it to the server. The server would perform some sort of computation on your query to transform it into a list of results, and then send the results back to you without ever knowing what it was that you searched for!

Many commonly used encryption schemes, like RSA, are *partially homomorphic* because they allow for some, but not all, operations to be performed on the encrypted texts [20]. However, for many decades it remained an unsolved problem as to whether there exists a *fully homomorphic* encryption scheme — a scheme where any operation can be performed on the encrypted texts [20]. In 2013, Craig Gentry's groundbreaking work [11,12] demonstrated that this "holy grail" of cryptography [17] does, in fact, exist.

This discovery and most derivative schemes [1–4, 6, 11–13, 22–24], obey the following format. The plaintext is encrypted with some sort of noise. As homomorphic operations are performed this noise grows until, after a certain amount of computation, it becomes so large that the ciphertext can no longer be decrypted. To mitigate this, a procedure called *bootstrapping* decrypts and reencrypts the text, entirely with homomorphic operations, resetting the noise to some small constant. By continuing to apply bootstrapping whenever necessary, an arbitrary number of operations can be performed. However, this comes with an immense computational overhead. So, in practice, many schemes are *somewhat homomorphic* [11,12]. These schemes do not bootstrap, which bounds the number of operations that can be performed before the text deteriorates.

In early work, homomorphic encryption was reported to be extremely slow, with single bit operations taking as long as half an hour to compute [12]. Recent advancements allow large circuits to be evaluated in seconds [1, 9]. But still, these circuits are limited to consist of only addition and multiplication operations. Although any function can theoretically be computed with these operations [11], it is not obvious how to do so efficiently [5, 7, 8, 15].

This paper investigates ways of computing these nonlinear functions including boolean comparisons and division under a particular model of homomorphic computing. In section 2, the specific constraints of this model are established. Section 3 provides a solution to this proposed problem and proves the solution is asymptotically optimal. Section 4 expands these results to include multivariate functions. Finally, section 5 provides experimental results and section 6 concludes with a discussion of some of the inherent limitations of homomorphic encryption revealed by these results.

2 A Model of Homomorphic Computing

The goal of this work is to describe methods of evaluating functions over the integers modulo t efficiently under homomorphic encryption. Setting aside the specifics of homomorphic schemes, the constraints of many of them can be generalized as follows [1–4, 6, 9, 11–13, 22–24]:

1. The available operations are addition and multiplication modulo the plaintext modulus, t.

2. Additions and scalar multiplications are relatively easy to compute, where as nonscalar multiplications are much harder to compute.

3. Additions and scalar multiplications do not increase the noise of ciphertexts by much, whereas nonscalar multiplications greatly increase the noise of ciphertexts.

Therefore, in order to evaluate functions efficiently, the computation must minimize the total number of nonscalar multiplications (reducing the total computation time), as well as the nonscalar multiplicative depth (reducing the overhead associated with noise growth).

2.1 Limitations of this model

Naturally, this model does not cover all cases as there are many homomorphic schemes, some of which have additional properties that can ease computation. For example, some configurations allow the individual bits of ciphertexts to be accessed and computed upon [5, 8, 15]. This make it possible to implement standard computer hardware algorithms that rely on bit shifting and bit logic to perform operations like evaluating inequalities and performing division.

Additionally, there are homomorphic schemes that support plaintext moduli so large that performing integer multiplications to the maximum multiplicative depth will not cause overflow. Under these schemes one can perform floating point operations by keeping track of a radix index for each integer and scaling down the decrypted result appropriately [5, 7].

While these alternatives are freeing, they currently come at enormous cost. In the bit method, the ciphertexts only encrypt single integers, making them orders of magnitude less efficient than schemes that use batch processing. As for ciphertexts that do not overflow, they must be immense in size, again sacrificing valuable performance.

3 Evaluating Single-Variable Functions

The key insight of my proposed solution is to form a more symbiotic relationship with the modulo operation that is built into the homomorphic addition and multiplication operations. I show that this modulus can be used as a tool for simplification, rather than a boundary that must be avoided, as in [5, 7].

Using this philosophy I will derive a method to compute any single variable homomorphic function. In section 3.1, I establish notation, in section 3.2 I discuss the conditions under which single variable functions are computable, and in 3.3 I discuss the algorithms that compute them and their complexity.

3.1 Notation

For clarity, the proofs and discussion to below use the following notation:

- A function is *computable* if it can be computed using only addition and multiplication operations modulo t.

- All homomorphically encrypted variables are represented in **bold**.

- All operations are assumed to be modulo t and equality "=" is used to represent congruence modulo t.

- $\delta : \mathbb{Z}/t\mathbb{Z} \to \{0, 1\}$ is the standard Dirac delta function, modulo t:

$$\delta(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x \neq 0 \end{cases} \tag{1}$$

3.2 Computability of Single-Variable Functions

In lemmas 1 and 2, I establish, through construction, that it is possible to compute any function with the given operations if the modulus, t, is prime. Lemmas 3 and 4 go on to enumerate the number distinct functions that exist as well as the number of ways that these functions can be computed. Theorem 1 concludes through a counting argument that all functions are computable if and only if the modulus t is prime, and furthermore that each of these functions has a unique polynomial that computes it.

Lemma 1. *If δ is computable, then any total function $f : \mathbb{Z}/t\mathbb{Z} \to \mathbb{Z}/t\mathbb{Z}$ is computable.*

Proof. Using the function δ, any total function $f : \mathbb{Z}/t\mathbb{Z} \to \mathbb{Z}/t\mathbb{Z}$ can be computed as follows:

$$f(x) = \sum_{k=0}^{t-1} f(k)\delta(x - k) \tag{2}$$

□

Lemma 2. *δ is computable if t is prime.*

Proof. If t is prime then δ can be computed as follows:

$$\delta(x) = 1 - x^{t-1} \tag{3}$$

By Fermat's Little Theorem [14], if $x \neq 0$ then $x^{t-1} = 1$ and so $\delta(x) = 0$. If $x = 0$, then $x^{t-1} = 0$ and so $\delta(x) = 1$. □

Lemma 3. *Exactly t^t distinct total functions $f : \mathbb{Z}/t\mathbb{Z} \to \mathbb{Z}/t\mathbb{Z}$ exist.*

Proof. Each of the $|\mathbb{Z}/t\mathbb{Z}|$ possible inputs to f corresponds to one of $|\mathbb{Z}/t\mathbb{Z}|$ possible outputs, therefore there are

$$|\mathbb{Z}/t\mathbb{Z}|^{|\mathbb{Z}/t\mathbb{Z}|} = t^t \tag{4}$$

distict functions that exist. □

Lemma 4. *At most $t^{\varphi(t)+1}$ distinct total functions $f : \mathbb{Z}/t\mathbb{Z} \to \mathbb{Z}/t\mathbb{Z}$ are computable.*

Proof. By Euler's Theorem [14], $x^{\varphi(t)+1} = x$, therefore any polynomial P can be reduced to the form

$$P(x) = \sum_{i=0}^{\varphi(t)} a_i x^i \tag{5}$$

This reduced polynomial has $\varphi(t)+1$ coefficients, each of which can take on one of $|\mathbb{Z}/t\mathbb{Z}| = t$ values, therefore $t^{\varphi(t)+1}$ such polynomials exist. Any computation involving only addition and multiplication modulo t is equivalent to a polynomial modulo t, therefore at most $t^{\varphi(t)+1}$ total functions can be computed. □

Theorem 1. *All possible functions $f : \mathbb{Z}/t\mathbb{Z} \to \mathbb{Z}/t\mathbb{Z}$ can be computed if and only if t is prime. Furthermore, if t is prime, then each function $f : \mathbb{Z}/t\mathbb{Z} \to \mathbb{Z}/t\mathbb{Z}$ is uniquely represented by a polynomial of degree less than t.*

Proof. If t is prime then δ can be computed by lemma 2. So, by lemma 1, all functions can be computed. If t is not prime then $\varphi(t)+1 < t$. So, by lemmas 3 and 4 there must exist some function that cannot be computed.

If t is prime, there are $t^{\varphi(t)+1} = t^t$ functions computable by polynomials and t^t distinct functions. Since no two distinct functions can be computed in the same way each function must be uniquely represented by exactly one polynomial. □

3.3 Complexity of Single-Variable Functions

Given the analysis in the previous section, consider the modulus t to be prime from this point forward. From the construction in lemma 1 and the definition of δ from lemma 2, we can calculate the minimal polynomial representing a function as follows:

$$P(x) = \sum_{k=0}^{t-1} f(k)\delta(x-k) \pmod{x^t - x} \tag{6}$$

Computing this polynomial takes $O(t^2 F)$ time where F is the complexity of $f(x)$. However, this computation occurs without any homomorphic overhead, so it can be considered negligible, for reasonable choices of f and t.

Theorem 1 proved that each function f has a unique polynomial that computes it. Therefore, there is no faster way to calculate $f(x)$ than by evaluating P at x. Paterson and Stockmeyer [19] proved that evaluating an arbitrary polynomial of degree t requires at least $\sqrt{t}$ nonscalar multiplications.

Algorithm 1 is a modification of Paterson and Stockmeyer's Algorithm B that maintains a logarithmic multiplicative depth. It requires approximately $3\sqrt{t}$ nonscalar multiplications and has a multiplicative depth of $O(\log t)$.

Therefore any single variable function can be computed homomorphically with $3\sqrt{t}$ nonscalar multiplications and a multiplicative depth of $O(\log t)$. This is asymptotically equal to the minimum number of nonscalar multiplications needed to compute an arbitrary single variable function.

Algorithm 1 Evaluate a single variable polynomial

function EVALUATEPOLYNOMIAL($\mathbf{x}$, P)

Let P be a polynomial $a_0 + a_1 x + \cdots + a_t x^t$ with $t = m^2 - 1$

$\mathbf{X}[0] \leftarrow 1$, $\mathbf{X}[1] \leftarrow \mathbf{x}$ ▷ $\mathbf{X}[i] = \mathbf{x}^i$

for $i = 2$ to m **do**
  $\mathbf{X}[i] = \mathbf{X}\left[\left\lfloor \frac{i}{2} \right\rfloor\right] \cdot \mathbf{X}\left[\left\lceil \frac{i}{2} \right\rceil\right]$ ▷ Compute powers $\mathbf{x}^2 \ldots \mathbf{x}^m$
end for

for $i = 2$ to $m - 1$ **do**
  $\mathbf{X}[im] = \mathbf{X}\left[\left\lfloor \frac{i}{2} \right\rfloor m\right] \cdot \mathbf{X}\left[\left\lceil \frac{i}{2} \right\rceil m\right]$ ▷ Compute powers $\mathbf{x}^{2m} \ldots \mathbf{x}^{(m-1)m}$
end for

$\mathbf{p} \leftarrow 0$ ▷ Compute the polynomial as
for $i = 0$ to $m - 1$ **do** ▷ $\sum_{i=0}^{m-1} \mathbf{x}^{im} \left(\sum_{j=0}^{m-1} a_{im+j} \mathbf{x}^j \right)$
  $\mathbf{q} \leftarrow 0$
  for $j = 0$ to $m - 1$ **do**
    $\mathbf{q} \leftarrow \mathbf{q} + a_{im+j} \cdot \mathbf{X}[j]$
  end for
  $\mathbf{p} \leftarrow \mathbf{p} + \mathbf{q} \cdot \mathbf{X}[im]$
end for
return p

end function

4 Multivariate Functions

Mimicking the single variable case, we can compute any bivariate function by evaluating the polynomial

$$P(x,y) = \sum_{k=0}^{t-1}\sum_{l=0}^{t-1} f(k,l)\delta(x-k)\delta(y-l) \pmod{x^t - x} \pmod{y^t - y} \quad (7)$$

This polynomial has $O(t^2)$ coefficients, and unfortunately the same tricks used in algorithm 1 cannot be easily applied to simplify the computation because of the intertwined variables.

So, evaluating the polynomial naively would take $O(t^2)$ multiplications, but section 4.1 below describes how many useful bivariate functions can be constructed by combining single-variable functions.

4.1 Linearly separable functions

Many functions, although not linear, can be evaluated as a combination of single variable functions and linear operations, giving them a complexity of $O(\sqrt{t})$. Table 4.1 demonstrates how division, logarithms, exponentiation and boolean comparisons are all linearly separable.

One thing to note about this table is that division, logarithms and exponentiation are all approximate. These functions and their subcomponents evaluate to real numbers, so they are not well defined over the integers. With some careful modification these approximations can be improved.

Consider the case of division. Resolution is lost when taking the logarithm, so to counteract this, we can inflate the output of the logarithmic functions to take advantage of their entire range, t. We also must account for the fact that the subtraction is done modulo t, so we will need to leave an extra bit to check if $y > x$. Finally, we must choose what should happen in the case of division by zero. Algorithm 2 demonstrates these modifications and chooses to return the numerator if the denominator is zero.

An almost identical algorithm can also be used to compute $\log_{\mathbf{b}} \mathbf{x}$. A homomorphic algorithm to compute exponentiation depends largely on the desired response when $\mathbf{x}^{\mathbf{y}} \geq t$. Discovering a more precise way to compute these functions is left to future work.

5 Experimental Results

The algorithms described in this paper have been implemented in C++ under the somewhat homomorphic encryption scheme YASHE (Yet Another Somewhat Homomorphic Encryption scheme) [1]. The library is built upon Victor Shoup's number theoretic library C++, NTL [21]. It is available along with the following experiments at `http://github.com/sportdeath/NonlinearSHE`.

The YASHE encryption is determined by the plaintext modulus t, the batch size β and the security parameter λ. Hidden parameters that determine the

Division	$\frac{\mathbf{x}}{\mathbf{y}} = \exp(\log \mathbf{x} - \log \mathbf{y})$
Logarithm	$\log_{\mathbf{b}} \mathbf{x} = \exp(\log\log \mathbf{x} - \log\log \mathbf{b})$
Exponentiation	$\mathbf{x}^{\mathbf{y}} = \exp(\mathbf{y} \log \mathbf{x})$
Equal To	$[\mathbf{x} = \mathbf{y}] = [\mathbf{x} - \mathbf{y} = 0]$
Greater Than	$\mathbf{a} = \left[\mathbf{x} \geq \frac{t-1}{2}\right]$ $\mathbf{b} = \left[\mathbf{y} \leq \frac{t-1}{2}\right]$ $\mathbf{c} = \left[(\mathbf{x} - \mathbf{y} \mod t) \leq \frac{t-1}{2}\right]$ $[\mathbf{x} \geq \mathbf{y}] = \mathbf{ab} + \mathbf{c}(\mathbf{a} + \mathbf{b} - 2\mathbf{ab})$

Table 1: Linearly separable functions.

batch size and security parameter are the cylotomic degree d and the ciphertext modulus q. Batches are made using the Chinese Remainder Theorem and the batch size is equal to the number of factors modulo t of the dth cyclotomic polynomial. For the system to be secure both d and q must be large. The degree of the dth cyclotomic polynomial is $n = \varphi(d)$, where φ is Euler's Toitient function. As per convention, I set the standard deviation of the ciphertext noise to be a constant; $\sigma_{err} = 8$.

I investigated YASHE parameters for both 8 and 16 bit integers. I use the Fermat primes $t = 2^8 + 1$ and $t = 2^{16} + 1$ as moduli for 8 and 16 bit integers respectively. Through manual search I have found values of d with maximal batch sizes that are large enough to be secure and provide a large multiplicative depth, while not being so large as to be computationally infeasible, shown in table 2.

For those batch sizes, I calculate the maximum value of $\log_2 q$ that guarantees λ bits of security with the following equation from [16]:

$$\log_2(q) \leq \min_{m>n} \frac{m^2 \cdot \log_2(\gamma(m)) + m \cdot \log_2\left(\sigma_{err}/\sqrt{\lambda \log(2)/\pi}\right)}{m - n} \tag{8}$$

I found the values of the minimal root Hermite factor $\gamma(m)$ by interpolating the

Algorithm 2 Approximate Homomorphic Division

function LOG($\mathbf{x}$)
  if $\mathbf{x} = 0$ **then**
    return 0
  else
    return $\left\lfloor \frac{t}{2} \log_t \mathbf{x} \right\rceil$
  end if
end function

function EXP($\mathbf{x}$)
  if $\mathbf{x} > \frac{t}{2}$ **then**
    return 0
  else
    return $\left\lfloor t^{\frac{2\mathbf{x}}{t}} \right\rfloor$
  end if
end function

function DIV($\mathbf{x}$, $\mathbf{y}$)
  return EXP(LOG($\mathbf{x}$) − LOG($\mathbf{y}$))
end function

t	d	Batch
$2^8 + 1$	22016	5376
$2^8 + 1$	66048	10752
$2^{16} + 1$	32768	16384
$2^{16} + 1$	65536	32768

Table 2: Feasible values of d and corresponding batch sizes.

values of $\gamma(m)$ found in Table 1 of [16]. Table 3 shows the derived values.

I have chosen to perform timing tests on the parameters shown in table 4, which are each optimized to have a multiplicative depth just large enough to support a particular operation. I have found that setting the radix, w, such that $\log_2 w \approx \frac{\log_2 q}{6}$ to be a good trade off between running time and multiplicative depth.

The timing results shown at the end of this paper in table 8 were computed on a 2012 Macbook Pro. NTL was not built with threading and all of the tests have been averaged over 10 trials. It is worth noting that computing *any* random function on the 8-bit inputs can be completed in as little as 4 milliseconds per input with the minimal number of layers. This function could be anything from division by a constant to a random mapping of inputs to outputs. Additionally, division of ciphertexts can be completed in 27 ms per input and comparison of ciphertexts can be completed in as little as 13 ms per input.

Compared to [5], this is roughly three times faster at computing comparisons.

max $\log_2 q$		Batch			
		5376	10752	16384	32768
λ	64	584.11	1195.53	992.54	1850.70
	80	437.49	1055.43	878.60	1627.87
	128	389.82	791.51	658.27	1218.44

Table 3: Number of bits in the ciphertext modulus for various security parameters and batch sizes.

#	t	$\log_2 q$	d	$\log_2 w$	λ	Operation
1	2^8+1	438	22016	74	≈ 80	Polynomial
2	2^8+1	546	22016	92	> 64	Inequality
3	2^8+1	930	66048	156	> 80	Division

Table 4: The chosen sets of timing parameters.

Moreover, the experiments in [5] were computed on only 5-bit integers rather than 8-bit integers. My division is 9 ms (66%) slower than their's, but again they were using smaller, 6-bit, integers. The biggest improvement however is that [5]'s algorithms were designed by hand for each function, so computing other more complicated functions is infeasible, whereas in my case it is very straight forward.

Some proof-of-concept image processing applications are also tested. These include transforming the color space of an image from RGB to YCbCr, performing a random color transformation on the inputs (an image filter), and mixing images together. Performing the color transformation involves performing fractional multiplications and can be computed in roughly 4 minutes on 70x70, 3 channel images shown in figure 1. The random color transformation involved computing a single variable function on a single channel, 8-bit image and completes in under 1 minute as shown in figure 2. Finally, taking the mean of images involved performing linear combination of fractional multiplications and completed in roughly 5 minutes as shown in figure 3.

6 Conclusion

In this paper I investigated the problem of computing nonlinear functions under the constraints of homomorphic encryption. I developed a computational model that describes the complexity of most homomorphic schemes. Under this model I showed that if the plaintext modulus t is prime, then any function of a single variable can be computed in $O(\sqrt{t})$ time. Moreover, many multivariate functions, like division and boolean operations, can be comprised of single variable functions to achieve the same complexity. My experimental results are both comparatively fast and demonstrate the correctness of my algorithms. Addi-

tionally, the results show that this method is capable of solving realistic image processing tasks that might be performed by a cloud computing service.

One issue that these results raise is that the complexity of evaluating a function is polynomial with respect to the *description* of the input, t. This makes this algorithm weakly polynomial. So, applying the techniques I used on 8-bit integers to 64-bit integers, or equivalently, doubles, would be almost 1 billion times slower. If homomorphic encryption is really intended to be used on real world data then it must be possible to do without this exponential overhead. This suggests that some part of the homomorphic computing model must be relaxed to allow for more than simply multiplication and addition modulo t.

As for the other proposed computational models [5, 7, 8, 15], it remains an open problem as to whether they allow for strongly polynomial evaluation.

7 Acknowledgements

I would like to thank my supervisor Sébastien Canard and the graduate students Marie Paindavoine, Donald Nokam Kuate, and Bastien Vialla for their generous feedback and support. Also, thank you to my coworkers and friends Yvan Raffle, Timothée Planchon, and Peter Mosely who made me feel at home in a far off land. À la prochaine!

References

[1] Joppe W. Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. Cryptology ePrint Archive, Report 2013/075, 2013. `http://eprint.iacr.org/2013/075`.

[2] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in cryptology–crypto 2012*, pages 868–886. Springer, 2012.

[3] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[4] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Annual cryptology conference*, pages 505–524. Springer, 2011.

[5] Gizem S. Cetin, Yarkin Doroz, Berk Sunar, and William J. Martin. Arithmetic using word-wise homomorphic encryption. Cryptology ePrint Archive, Report 2015/1195, 2015. `http://eprint.iacr.org/2015/1195`.

[6] Jean-Sébastien Coron, Avradip Mandal, David Naccache, and Mehdi Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In *Annual Cryptology Conference*, pages 487–504. Springer, 2011.

[7] Anamaria Costache, Nigel P Smart, Srinivas Vivek, and Adrian Waller. Fixed-point arithmetic in she schemes. In *International Conference on Selected Areas in Cryptography*, pages 401–422. Springer, 2016.

[8] Yarkın Doröz, Yin Hu, and Berk Sunar. Homomorphic aes evaluation using the modified ltv scheme. *Designs, Codes and Cryptography*, 80(2):333–358, 2016.

[9] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[10] Juan Garay, Berry Schoenmakers, and José Villegas. Practical and secure solutions for integer comparison. In *International Workshop on Public Key Cryptography*, pages 330–342. Springer, 2007.

[11] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[12] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. Cryptology ePrint Archive, Report 2010/520, 2010. `http://eprint.iacr.org/2010/520`.

[13] Craig Gentry and Shai Halevi. Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 107–109. IEEE, 2011.

[14] Godfrey Harold Hardy, Edward Maitland Wright, et al. *An introduction to the theory of numbers*. Oxford university press, 1979.

[15] Kristin Lauter, Adriana López-Alt, and Michael Naehrig. Private computation on encrypted genomic data. In *International Conference on Cryptology and Information Security in Latin America*, pages 3–27. Springer, 2014.

[16] Tancrède Lepoint and Michael Naehrig. A comparison of the homomorphic encryption schemes fv and yashe. Cryptology ePrint Archive, Report 2014/062, 2014. `http://eprint.iacr.org/2014/062`.

[17] Daniele Micciancio. A first glimpse of cryptography's holy grail. *Communications of the ACM*, 53(3):96–96, 2010.

[18] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 113–124. ACM, 2011.

[19] Michael S. Paterson and Larry J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 1973.

[20] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[21] Victor Shoup. NTL: Number Theory Library, 2016. `shoup.net/ntl/`.

[22] Nigel P Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *International Workshop on Public Key Cryptography*, pages 420–443. Springer, 2010.

[23] Damien Stehlé and Ron Steinfeld. Faster fully homomorphic encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 377–394. Springer, 2010.

[24] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 24–43. Springer, 2010.

8 Appendix

Figure 1: A homomorphic transformation of Mona Lisa from RGB to YCbCr preserves visual similarity. The Y, Cb, and Cr channels are displayed in the middle

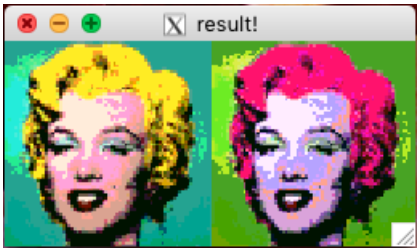

Figure 2: The input (left) and output (right) of an arbitrary homomorphic color transformation.

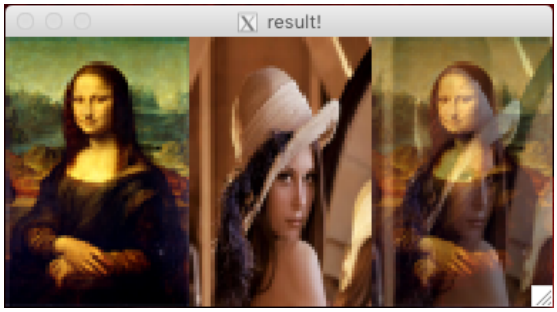

Figure 3: A linear mixture of Mona Lisa and Lena, computed homomorphically.

#	Time/Batch	Time/Operation
Encryption		
1	6659.2 ms	1.239 ms
2	6648.9 ms	1.237 ms
3	38 263.2 ms	3.559 ms
Decryption		
1	9938.4 ms	1.849 ms
2	9958.3 ms	1.852 ms
3	39 974.1 ms	3.718 ms
Addition with Constant		
1	18.4 µs	3.423 ns
2	16.6 µs	3.088 ns
3	27.2 µs	2.530 ns
Multiplication with Constant		
1	1974.2 µs	367.2 ns
2	2186.9 µs	406.8 ns
3	5496.7 µs	511.1 ns
Addition of Ciphertexts		
1	417.6 µs	77.679 ns
2	501.6 µs	93.304 ns
3	6099.4 µs	567.281 ns
Multiplication of Ciphertexts		
. 1	462.0 ms	85.932 µs
2	560.7 ms	104.288 µs
3	2306.4 ms	214.511 µs
Random Single Variable Function		
1	20 248.2 ms	3.766 ms
2	25 029.7 ms	4.656 ms
3	91 361.1 ms	8.497 ms
Comparison of Ciphertexts		
2	71 071.4 ms	13.220 ms
3	300 576 ms	27.955 ms
Division of Ciphertexts		
3	294 826 ms	27.688 ms

Table 5: Timing results from the parameters in table 4.