Algorithms for Evaluating Nonlinear Functions Under Homomorphic Encryption

Trevor Henderson

August 30, 2016

Abstract

This paper looks at methods of computing nonlinear functions efficiently under the constraints of homomorphic computation — 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 an arbitrary single variable function. I then show that many multivariate functions, such as division and boolean comparisons can be evaluated using $O(\sqrt{t})$ multiplications. The multiplicative depth for both algorithms is $O(\log t)$.

Experimental results utilizing an implementation of the somewhat homomorphic scheme YASHE boast secure homomorphic evaluations of arbitrary 8-bit single variable functions in less than 4 milliseconds amortized on a 2012 MacBook Pro.

1 Homomorphic Encryption Overview

Homomorphic encryption let's you encrypt messages and then do math on the encrypted messages.

While there are many different homomorphic schemes, all of them rely on encrypting the plaintext with some sort of noise. As operations are performed on the ciphertexts this noise grows until, after a certain amount of computation, it becomes so large that the ciphertext can no longer be decrypted. A procedure called bootstrapping homomorphically decrypts a message, resetting its noise to some small constant. With bootstrapping, circuits of arbitrary depth can be evaluated. However bootstrapping has an immense computational overhead, so in practice many schemes are somewhat homomorphic, in that there computational depth is bounded [3].

Homomorphic encryption began as a very slow process where single bit operations could take as much as half an hour to compute [4]. Now however we can evaluate large circuits in seconds. But these circuits are very limited, as they can only consist of addition and multiplication. Many operations used in a standard computing model like equality testing, bit shifting, floating point multiplication, and division cannot be computed naturally with just those two operations. Therefore this paper looks at efficient ways of computing such functions.

2 Problem Description

The goal is to evaluate functions $f : \mathbb{Z}/t\mathbb{Z} \to \mathbb{Z}/t\mathbb{Z}$ efficiently under homomorphic encryption. Setting aside the specifics of homomorphic schemes, we describe the constraints of this problem as follows.

1. We only have access to addition and multiplication operations 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 — to reduce the total computation time — as well as the nonscalar multiplicative depth — to reduce the computational overhead associated with noise growth.

2.1 Limitations of the Problem Description

Naturally, this description cannot cover all cases as there are many homomorphic schemes, some with additional properties that can ease computation. For example, some homomorphic schemes allow the individual bits of ciphertexts to be accessed and computed upon. 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 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 [2].

While these alternatives and others are freeing, they tend to come at a large cost. Oftentimes ciphertexts will only encrypt single integers, losing orders of magnitude of computation time that could be gained with batch processing. The ciphertexts can become bloated to allow for room for overflow, increasing their size and the time it takes to perform computations on them immensely.

Additionally, from a purely aesthetic standpoint, adding other helper variables like a radix, takes away from some of the magic of homomorphic computation. It is far less amazing to unbox something half finished, than it is to have a solution appear all at once and ready to go.

Still, these methods are useful in areas where complicated homomorphic functions must be calculated on only a few, large inputs but I will not discuss them further.

3 Solution Description

My solution to the problem works to form a more symbiotic relationship with the modulo operation that is built into the homomorphic addition and multiplication operations. I use it as a tool for reduction rather than a boundary to be avoided.

Using this philosophy I derive a method to compute any single variable homomorphic function. I prove this method is an asymptotic lower bound. Then I investigate several possible methods of computing multivariate functions.

This method can very efficiently perform batch operations on relatively small (8 or 16-bit) integers. This makes it very useful for problems like image processing.

3.1 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 modulo t.

4 Single Variable Functions

Let us define the function $\delta : \mathbb{Z}/t\mathbb{Z} \to \{0, 1\}$ as follows:

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

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. *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

$$|\mathbb{Z}/t\mathbb{Z}|^{|\mathbb{Z}/t\mathbb{Z}|} = t^t \tag{3}$$

total functions that exist. □

Lemma 3. *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, $x^{\varphi(t)+1} = x$, therefore any polynomial $P : \mathbb{Z}/t\mathbb{Z} \to \mathbb{Z}/t\mathbb{Z}$ can be reduced to the form

$$P(x) = \sum_{i=0}^{\varphi(t)} a_i x^i \tag{4}$$

This reduced polynomial has $\varphi(t) + 1$ coefficients, each of which 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. □

Lemma 4. *δ is computable if t is prime.*

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

$$\delta(x) = 1 - x^{t-1} \tag{5}$$

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

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.*

Proof. If t is prime then δ can be computed by Lemma 4 so by Lemma 1, all functions can be computed. If t is not prime then $\varphi(t) + 1 < t$. So by Lemmas 2 and 3 there exist some functions which cannot be computed. □

Theorem 2. *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. There are $t^{\phi(t)+1} = t^t$ distinct polynomials and t^t total functions, therefore every function must be uniquely represented by exactly one polynomial. □

From this point on we will consider the modulus t to be prime. We can calculate the minimal polynomial representing a function as follows:

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

This polynomial can be calculated in the clear so it has negligible impact on running time. As this polynomial is the only polynomial with degree less than t equivalent to f, there is no faster way to calculate $f(x)$ than by evaluating P at x. Paterson and Stockmeyer [6] 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 which maintains a logarithmic multiplicative depth. It requires approximately $3\sqrt{t}$ nonscalar multiplications and has a multiplicative depth of approximately $\log_2 t$.

Algorithm 1 Evaluate a single variable polynomial

```
function EvaluatePolynomial($\mathbf{x}$, $P$)

    Let $P$ be a polynomial $a_0 + a_1x + \cdots + a_tx^t$ 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 $\mathbf{p}$

end function
```

Therefore any single variable function can be computed homomorphically with no more than $3\sqrt{t}$ nonscalar multiplications and a multiplicative depth of no more than $\log_2 t$. This is asymptotically equal to the minimum amount of nonscalar multiplications needed to compute an arbitrary single variable function.

5 Multivariate Functions

5.1 Brute Force

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

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

This polynomial has $O(t^2)$ coefficients. A trivial algorithm for computing this polynomial is shown in Algorithm 2.

Algorithm 2 Evaluate a multivariate polynomial

```
function EvaluatePolynomial($\mathbf{x}$, $\mathbf{y}$, $P$)

    Let $P$ be a polynomial $a_{0,0} + \cdots + a_{i,j}x^iy^j + \cdots + a_{t,t}x^ty^t$

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

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

    $p \leftarrow 0$
    for $i = 0$ to $t-1$ do
        for $j = 0$ to $t-1$ do
            $p \leftarrow p + a_{ij}\mathbf{X}[i]\mathbf{Y}[j]$    ▷ Multiply coefficients and variables
        end for
    end for
    return p

end function
```

This algorithm requires $O(t^2)$ multiplications and has a multiplicative depth of $\log_2 t + 1$. The multivariate case is indeed much slower than the univariate case as the intertwined variables make it difficult to decrease the total number of nonscalar multiplications without greatly increasing the nonscalar multiplicative depth. Additionally, for functions with even more variables, this method quickly suffers the curse of dimensionality. For extremely random two dimensional functions brute force might be the only option, however as we will see most practical functions can be evaluated much faster.

5.2 Linearly separable functions

Many functions, although not linear, can be evaluated as a combination of single variable functions and linear operations. For example the following common functions are linearly separable:

- Equal To:

$$[\mathbf{x} = \mathbf{y}] = [\mathbf{x} - \mathbf{y} = 0] \tag{8}$$

- Greater Than:
  Let

$$\mathbf{a} = \left[\mathbf{x} \geq \frac{t-1}{2}\right] \tag{9}$$

$$\mathbf{b} = \left[\mathbf{y} \leq \frac{t-1}{2}\right] \tag{10}$$

$$\mathbf{c} = \left[(\mathbf{x} - \mathbf{y} \mod t) \leq \frac{t-1}{2}\right] \tag{11}$$

  Then

$$[\mathbf{x} \geq \mathbf{y}] = \mathbf{ab} + \mathbf{c}(\mathbf{a} + \mathbf{b} - 2\mathbf{ab}) \tag{12}$$

- Division:

$$\frac{\mathbf{x}}{\mathbf{y}} = \exp(\log \mathbf{x} - \log \mathbf{y}) \tag{13}$$

- Logarithm:

$$\log_{\mathbf{b}} \mathbf{x} = \exp(\log \log \mathbf{x} - \log \log \mathbf{b}) \tag{14}$$

- Exponentiation:

$$\mathbf{x}^{\mathbf{y}} = \exp(\mathbf{y} \log \mathbf{x}) \tag{15}$$

The boolean expressions evaluate to either integers, either 0 or 1. However, division, logarithm and exponentiation algorithms evaluate to real numbers, so they must be modified before they can be used homomorphically. Consider the case of division. Since we can only perform integer operations, we lose resolution when taking logarithms. To counteract this we can inflate the output of the logarithmic functions to take advantage of the entire range. 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 the what should happen in the case of division by zero. Algorithm 3 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 $\mathbf{x}^{\mathbf{y}} \geq t$. One thing to note is that division, logarithms and exponentiation computed this way will be approximate. Improving these approximations is discussed in the next section.

Algorithm 3 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 $\left\lfloor t^{\frac{2\mathbf{x}}{t}} \right\rceil$
    end if
end function

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

6 Future work

6.1 Optimization of linearly separable functions

In certain applications an approximate algorithms will not be acceptable. An interesting problem would be to see if these approximations could be improved upon automatically using machine learning. For example, the optimization problem for division could be formulated as follows:

Problem 1. *Given the form $F(x, y) = f_1(f_2(x) + f_3(y))$, determine f_1, f_2, f_3 that minimize*

$$\sum_{0 \le x, y < t} \left(F(x, y) - \left\lfloor \frac{x}{y} \right\rceil \right)^2 \tag{16}$$

Here f_1, f_2, and f_3 could be considered arrays of size t with integers in $\mathbb{Z}/t\mathbb{Z}$, therefore the search space is finite albeit very large. Simple hill climbing on these functions from the assignments produced by Algorithm 3 can make mild improvements. However my attempts to use other methods including simulated annealing and genetic algorithms have not yet been successful.

6.2 Non linearly separable functions

In order to evaluate functions that are not naturally separable, machine learning techniques could be used to find approximate them. Neural networks for example could be implemented to approximate nonlinear functions. Neural networks rely on three operations: addition, multiplication and the sigmoid function. The sigmoid function is a function of a single variable therefore it could be implemented homomorphically using the methods described above.

7 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 [7].

The system is determined by the plaintext modulus t, the batch size β and the security parameter λ. Hidden parameters for the YASHE scheme that determine the batch size and security parameter and the cylotomic degree d, and the ciphertext modulus q. Batches are made using the Chinese Remainder Theorem and 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. We set the standard deviation of the ciphertext noise $\sigma_{err} = 8$ to be constant.

I have 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 large enough to be secure and provide a large multiplicative depth, while not being so large as to be computationally infeasible.

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

For those batch sizes, we calculate the maximum value of $\log_2 q$ that guarantees λ bits of security with the following equation from [5]:

$$\log_2(q) \leq \min_{m>n} \frac{m^2 \cdot \log_2(\gamma(m)) + m \cdot \log_2\left(\sigma_{err}/\sqrt{\lambda \log(2)/\pi}\right)}{m-n} \tag{17}$$

I found the values of the minimal root Hermite factor $\gamma(m)$ by interpolating the values of $\gamma(m)$ found in Table 1 of [5] The following is a table of the derived values:

max		Batch			
$\log_2 q$		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

I have chosen to perform timing tests on the following sets of parameters 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.

#	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

The following timing results were computed on a 2012 Macbook Pro. NTL was not built with threading. All of the tests have been averaged over 10 trials.

#	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

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] 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`.

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

[4] 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`.

[5] Tancrde 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`.

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

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