CSE 441/541: Cryptography and Applications

Research Project:

Time Complexities of Computations Done on Homomorphic Encryption Algorithms

Eric Binnendyk and Brenton Candelaria

New Mexico Institute of Mining and Technology
Socorro, NM 87801, USA

Date: December 3, 2021

Abstract

We survey and investigate the most efficient known ways to implement various common algorithms, including Boolean operations, addition, sorting a list, and simulating Turing machines, in a homomorphic encryption scheme developed by Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. We provide upper bounds on the complexity required to compute these operations.

1 Introduction

In the modern era of cybersecurity, many existing cryptographic algorithms are touted for having high entropy and for being highly sensitive to slight changes in both the key and plaintext. This would mean that the ciphertext in transit is sensitive, as either of these factors and can result in unusable or incorrect plaintext at the output should it be tampered with. Thus, for operations to be done on encrypted data, it is first necessary to decrypt said data and then perform the operation before re-encrypting it. A **homomorphic encryption scheme**, however, allows one to perform operations on encrypted values without first decrypting them. A **fully homomorphic encryption** (FHE) scheme allows for arbitrary computation to be performed on the encrypted data, that is, it supports the computation of a functionally complete set of operations [1].

Despite being touted as the future of cryptography, there is no known FHE scheme fast enough to be efficient for anything other than small-scale problems.[4] In this paper, we will investigate and explain how to do computations of sorting, binary search, and primality testing on homomorphically encrypted data, and why it is not fast enough yet to achieve widespread use.

If FHE becomes viable, it is predicted to have many specialized applications, including consumer privacy in targeted advertising, outsourcing storage and computations of confidential information, private queries to a database, and a simple protocol for zero-knowledge proofs. [9]

2 Background

2.1 Homomorphic encryption schemes

A homomorphic encryption scheme consists of the following functions:

- $Setup(\lambda)$ – Returns the algorithm parameters corresponding to the security parameter λ
- $SecretKeyGen(params)$ – generates the secret key
- $PublicKeyGen(params, K_s)$ – generates the public key
  - In some schemes there is a single algorithm $KeyGen$ that generates both keys [8]
- $Encrypt(K_p, M)$ – encrypts using public key
- $Decrypt(K_s, C)$ – decrypts ciphertext with secret key
- $Eval(K_p, f, C_1, \ldots, C_k)$ – evaluates a value such that $Decrypt(K_s, Eval(K_p, f, C_1, \ldots, C_k)) = f(M_1, \ldots, M_k)$, where $M_1, \ldots, M_k$ are the plaintexts of $C_1, \ldots, C_k$

The function f is stored as an **arithmetic circuit**, a computational model which is a **directed acyclic graph** with inputs, outputs, and nodes representing operations. Each operation is either addition or multiplication in some ring R. Addition and multiplication nodes have a **fan-in** of two: they can only have two inputs but can be used as many times as possible.

A **partially homomorphic encryption scheme** supports just one kind of operation on the encrypted data (addition or multiplication). This kind of scheme does not allow for universal computation [11].

A **fully homomorphic encryption scheme** is one that can simulate both addition and multiplication on the plaintext. The minimum theoretical requirements are addition and multiplication modulo 2 on 0, 1. This is sufficient to simulate any Boolean circuit and thus allows for universal computation, as shown below.

In fully homomorphic encryption, operations can theoretically be performed on the ciphertext an infinite number of times. One of the biggest challenges is to reduce "noise" that builds up over time in the ciphertext as more calculations get performed. This can be achieved via a technique called *bootstrapping*, which will be discussed more later. [4]

2.2 History and related work

After the inception of RSA by Rivest, Adleman, and Shamir, it was discovered that it is in fact possible to compute the product of plaintexts with only the corresponding ciphertexts and the public key. This notion was coined a "privacy homomorphism" by Rivest, Adleman, and Dertouzos [13]. The authors then asked questions regarding such systems, such as for what algabraic systems do useful privacy homomorphisms exist, what are the capabilities of such cryptographic systems, and what are their limitations?

There have been several examples of homomorphic encryption. Craig Gentry's scheme was foundational in being the first scheme to use **bootstrapping**, a term for encryption schemes that are capable of evaluating their own decryption circuit homomorphically. The proposed algabraic system for this bootstrapping method was over ideal lattices, and by definition, utilizes circuits that are $O(\lambda)$, where λ is the security parameter [8].

Later, Brakerski, Gentry, and Vaikunatanathan developed a vastly different approach to fully homomorphic encryption. In their paper, they had proposed a scheme that was $O(\lambda \cdot L^3)$ (L being the depth of the circuit) without bootstrapping, and a $O(\lambda^2)$ scheme with bootstrapping [3]. Both of these schemes are what we will examine in this paper.

That is not to say there exist no other homomorphic schemes, or even other fully homomorphic schemes. Parmer et al. studied and compared the capabilities of several known schemes in 2014 [12], including the bootstrapping scheme devised by Brakerski et al.

2.3 Mathematical background of Brakerski et al's leveled FHE scheme

An **ideal** of a ring R is a subset $I \subseteq R$ that is a subgroup under addition such that I is closed under multiplication by any $r \in R$. For an ideal I, the sets $r + I$ for $r \in R$ are **residue classes** of I [8].

The ring $\mathbb{Z}[x]$ is composed of polynomials with integer coefficients. The **quotient ring** $\mathbb{Z}[x]/P(x)$ is the set of integer polynomials modulo some irreducible polynomial $P(x)$. In the BGV scheme [3], this polynomial is of the form $x^{2^k} + 1$. This ring can be modeled by a lattice of points with integer coefficients. The set $n\mathbb{Z}[x]/P(x)$ is an ideal of $\mathbb{Z}[x]/P(x)$ consisting of polynomials mod $P(x)$ whose coefficients are divisible by n. In Brakerski et al's scheme, there is a canonical way to choose one element from each coset $r + n\mathbb{Z}[x]/P(x)$ to perform computations mod $n\mathbb{Z}[x]/P(x)$.

Ring learning with errors (RLWE) problem: Brakerski et al use this problem as the basis of their homomorphic encryption schemes. The problem goes as follows:
Given an integer $q > 2$ dimension n, and distribution χ over R, and with access to samples from A_χ (defined as the distribution $\{(a_i, a_i \cdot s + e) | a_i, s \in R_q, e \sim \chi\}$), distinguish A_χ from the uniform distribution over $R_q \times R_q$. This infeasibility of this problem is the central assumption for the hardness of the FHE scheme.

2.4 Limitations of homomorphic encryption schemes

All homomorphic encryption schemes known to date are too slow to have gained widespread use[cite]. In addition, homomorphic encryption schemes are not secure against **adaptively chosen ciphertext attacks**, meaning attacks where

the decryption oracle can be queried on ciphertext which is chosen based on the decryption of previous ciphertexts. In some sense, this is because homomorphic encryption is "learnable". [8]

As an example, if an encryption of "1" is known, then an encryption of any other number can be generated by adding a number of 1's together, or by adding powers of 2, achieved as 1 + 1. Thus, homomorphically encrypted data is unsuitable to be used for authentication. On the other hand, there are multiple possible encryptions of the same plaintext, so the encryption is resistant to a frequency analysis attack that may work against an ECB (electronic code book) cipher.

3 Complexity of some operations

Universal computation can be achieved by simulating Boolean circuits from arithmetic circuits over any ring R using the following equalities:
Let $x, y \in \{0, 1\}$, where 0 and 1 are the additive and multiplicative identity of R. Identify 0 with False and 1 with True. Then:

$$x \wedge y = xy \qquad \text{Complexity: One gate, depth one}$$
$$x \vee y = x + y - xy \qquad \text{Complexity: Three gates, depth three}$$
$$\neg x = 1 - x = 1 + (-1)x \qquad \text{Complexity: Two gates, depth two}$$

In fact, all Boolean functions can be written as polynomials over the field $\mathbb{F}_2$, where no variable is raised to a power higher than 1 (because $x^2 = x$ in $\mathbb{F}_2$). This means that a Boolean function of n variables can be achieved with at most 2^n additions and $n \cdot 2^n$ multiplications forming n groups. The circuit size is $O(n \cdot 2^n)$ gates and $O(n \log(n))$ depth using binary trees to simulate unlimited fanin. (We rediscovered this binary tree method ourselves but it is likely a well-known result in circuit complexity literature.)

3.1 Sorting numeric inputs

Here we describe how to sort a list of m integers from 0 to n. We implement it using modular arithmetic over the field $\mathbb{Z}_q$. This is a subring of the ring $\mathbb{Z}[x]/q\mathbb{Z}[x]$, and thus the same computation will work with this field. Suppose two elements, $a, b \in \mathbb{Z}_q$ are chosen, then:

First, we require a prime modulus q that is greater than $2n$. Because there must be a prime between $2n$ and $3n$, $q = O(n)$.

3.1.1 Greater than function

We describe how to make a "greater than" function that is valid for integers from 0 to n, encoded as integers mod p, where $n < p$.
Define $IsZero(x)$:

$$IsZero(x) = 1 - x^{p-1} \qquad \text{Complexity: } O(\log p), \text{ depth: } O(\log p)$$

See Fig. 1 for an illustration of this circuit.

$GreaterThan(a, b)$ is implemented as:

$$IsZero(a-b-1) \vee IsZero(a-b-2) \vee IsZero(a-b-3) \vee \cdots \vee IsZero(a-b-n)$$

Complexity: $O(n \log q) = O(n \log n)$ gates, depth $O(n \log q) + O(\log n) = O(n \log n)$

See Fig. 2 for an illustration of this circuit.

The runtime is $O(\lambda(n \log n)^4)$ in the non-bootstrapping case, $O(\lambda^2 n \log n)$ in the bootstrapping case.

Note that this is the part of the design that only works if $n < 2q$, because otherwise $IsZero(a-b-n)$ might be satisfied even if $a \leq b$.

3.1.2 Condition function

$Cond(x, a, b)$ is a function that returns b if $x = 0$, or a if $x = 1$, similar to the built-in cond function in Lisp. It is implemented as:

$$b + ax + (-1)bx$$

Complexity: Five gates, depth four

See Fig. 3 for an illustration of this circuit.

3.1.3 Ordering function

$Order2(a, b)$ takes two elements and orders them. It is implemented as:

1: **function** ORDER2(a, b)
2: $x_1 \leftarrow GreaterThan(b, a)$
3: $s \leftarrow Cond(x_1, a, b)$
4: $x_2 \leftarrow GreaterThan(a, b)$
5: $l \leftarrow Cond(x_2, b, a)$ **return** (s, l)
See Fig. 4 for an illustration of this circuit.
6:

Complexity: $O(n \log(n))$ gates

3.1.4 Sorting function

Sorting can be implemented as follows:

1: **function** SORT($x_1, x_2, ..., x_n$)
2: **for** i=2..n **do**
3: **for** j=i - 1 down to 1 **do**
4: $x_j, x_{j+1} \leftarrow Order2(x_j, x_{j+1})$

See Fig. 5 for an illustration of this circuit.

Each of the m elements are evaluated pairwise using $Order2()$ in a manner much like insertion sort. These operations can be parallelized so the circuit has a depth of $O(m)$ in terms of other circuits, and $m(m-1)/2$ $Order2$ gates. This yields a complexity of $O(m^2 n \log(p))$, a cost of $O(m^2 n \log(p)\lambda^2)$ in the bootstrapping case, and a cost of $O(m^8 n \log(n)^3 \log(p)\lambda)$ in the non-bootstrapping case.

3.2 Sorting inputs stored in binary

Cetin et al [6] describes an alternative method of sorting, where the elements of the list are encrypted bit-by-bit. Their method involves integers stored as encrypted binary strings of equal length. It should be noted that because the FHE algorithm is randomized, a single bit can be encrypted in multiple ways. To test for equality in this scheme, their circuit multiplies the result of XOR'ing each bit pair and that result with 1. Trivially, this operation can only result in 1 when the bits in both strings are identical. To evaluate less-than, they consider the product sum of each string and evaluate products utilizing a binary tree over $l+1$ elements for l-bit binary strings. The advantage of this method is that the less-than function for binary strings takes $O(\log n + 1)$ gates, not $O(n)$ as in the numeric case.

They also define a swap function not unlike the $Order2()$ function we propose. Their implementation of swap has depth $O(\log(n+1)+1)$ for binary strings of length n. In the same paper, they propose much more clever sorting algorithms such as direct sort, which is able to sort an array of length N in circuit depth $O(\log N \log l)$. In their paper, they demonstrate that this sorting method and another they proposed, greedy sort, vastly reduce the multiplicative depth of circuits when compared to other known algorithms such as Bubble sort or Merge sort.

An indirect result from the Cetin paper demonstrates that some algorithms that are efficient in the plaintext space may not necessarily be efficient in the ciphertext space of a fully homomorphic scheme. This is because the complexity and security parameter of a fully homomorphic scheme is closely related to the multiplicative depth of circuits the scheme evaluates. Thus, it is necessary to optimize circuits for minimum depth to obtain useful and efficient computation.

3.2.1 Other problems over binary ciphertext

Certain unique qualities arise from ciphertext generated from binary strings. Since many circuits can be evaluated bitwise over the ciphertext, it becomes possible to design certain protocols or algorithms that provide for efficient computation of comparisons or other meaningful operations. An example is the Millionaire's problem, whereby two parties wish to determine who possesses more assets without openly disclosing the values of these assets.

Brenner et al propose a number of protocols and algorithms over a fully homomorphic scheme that solve this problem and others [4]. Namely, their solution to the Millionaire's problem involves the addition of the one's compliment of the binary strings of each party, then observation of the sign bit. They also propose a method for searching a list of encrypted data bitwise that is constant in circuit depth with respect to the size of each encrypted entry.

3.3 Complexity of binary search

Binary search takes three arguments, a sorted array A of homomorphically encrypted data, a number x to search for, and the public key K_p. This scheme assumes that the function executor has access to oracles that can decrypt the Boolean outputs of $GreaterThan$ and $IsEqual$, or otherwise has memorized all possible encryptions of 0 and 1. Suppose additionally that the size of the array is readily available via $A.size$.

3.3.1 Equality function

The function $IsEqual(a, b)$ checks whether its inputs are encryptions of the same value in $\mathbb{Z}_q$. It returns a value that is an encryption of 0 (false) or 1 (true). This function can easily be derived from $IsZero$ above by checking if the difference is 0:

$$IsEqual(a, b) = 1 - (a - b)^{q-1} \qquad \text{Complexity: } O(\log q) \text{ gates, depth } O(\log q)$$

The runtime is $O(\lambda(\log n)^4)$ in the non-bootstrapping case, $O(\lambda^2 \log n)$ in the bootstrapping case.

$BinSearch(A, x, K_p)$ can be implemented as:

```
1:  function BinSearch(A, x, K_p)
2:      l ← 0
3:      h ← A.size
4:      c ← ⌊h/2⌋
5:      q ← Encrypt(K_p, x)
6:      while h ≠ l do
7:          if GreaterThan(q, A[c]) then
8:              l ← c
9:          else
10:             if IsEqual(q, A[c]) then return x
11:             h ← c
12:         c ← l + ⌊(h−l)/2⌋
        return null
```

Something notable about the binary search algorithm is that most variable assignments do not rely on the homomorphic scheme at all, and only the conditional checks need to deal with the encrypted data. The while loop at line 6 runs a total of $\log m$ times, where $m = A.size$. This means the check at line 7 is always run $\log m$ times. At worst case, the check at line 10 is run $\log m$ times as well, meaning the complexity of the entire algorithm is $O(\lambda(n \log n)^4 \log m)$ (non-bootstrapping) or $O(\lambda^2 n \log n \log m)$ (bootstrapping).

3.4 Complexity of primality testing

This is based on the Miller-Rabin algorithm. It takes in a value x and a witness $w < x$. If it returns true, x may or may not be prime, but if it returns false, x is definitely composite.

```
1: function IsPrime(x, w)        ▷ x and w are stored as binary strings, x is the candidate prime, w is a witness
2:     ℓ ← len(x)
3:     y ← x − 1
4:     ▷ Take w^y by squaring repeatedly, and check whether any power is 1 when the previous power is not 1 or y during each squaring step
5:     pow ← 1
6:     prev_pow ← 1
7:     maybe_prime ← True
8:     for i ← 1..ℓ do
9:         prev_pow ← pow
10:        pow ← pow^2 · cond(y[i], pow, 1) mod x
11:        maybe_prime ← maybe_prime ∧ ¬(y[i] = 0 ∧ pow = 1 ∧ prev_pow ≠ 1 ∧ prev_pow ≠ y)
12:    maybe_prime ← maybe_prime ∧ (pow = 1)        ▷ The final power needs to be 1 due to Fermat's little theorem
13: return maybe_prime
```

Note that this algorithm uses bounded loops and does not have conditional statements. This makes it easy to implement as a circuit, because the same path of execution is followed no matter the input values.

The complexity of the most efficient *practical* circuit for multiplication is $O(n^2)$. (There are multiplication circuits with $O(n \log n 2^{\log^* n})$ gates, but they are very large for small values of n.) [2] We were unable to find a good reference for the complexity of modulo circuits. If the complexity is $M(n)$, the number of gates is $O(n^3 M(n))$ and the depth is $O(n)$, so the complexity in BGV is $O(\lambda n^6 M(n))$ without bootstrapping and $O(\lambda^2 n^3 M(n))$ with bootstrapping.

3.5 Complexity of polynomial evaluation

Another potentially useful algorithm in homomorphic encryption is the evaluation of an arbitrary polynomial for some value, x. Neither of the polynomial nor the point x need to be explicitly known (i.e.: can be encrypted). Suppose the polynomial is passed as an n dimensional vector for an n^{th} degree polynomial with its entries being the respective coefficients (so the 1^{st} entry is the constant coefficient), then the polynomial can be evaluated as follows:

```
1: function EvalPoly(A, x)
2:     z ← 0
3:     for i ← 1, i < A.length do
4:         z ← z + v[i]
5:         z ← z ∗ x
6:     z ← z + v[A.length] return z
```

Notice that the runtime is proportional to the degree of the polynomial from line 3. This means that the runtime of the algorithm is $O(\lambda n^3)$ for the non-bootstrapping case, and $O(\lambda^2 n)$ for the bootstrapping case.

3.6 Complexity of arbitrary algorithms

Any algorithm that takes $O(f(n))$ time on a Turing machine can be converted into a Boolean (and thus arithmetic) circuit with $O(f(n)^2)$ gates and a depth of $O(f(n))$ ([5], theorem 1), as shown in Fig. 6. The homomorphic computation has a

time complexity of $O(\lambda f(n)^5)$ without bootstrapping, or $O(\lambda f(n)^2)$ with bootstrapping. This result means that universal computation is possible over the ciphertext space of a fully homomorphic scheme, and that the circuit depth is linear with respect to the algorithm's runtime complexity.

4 Conclusion

Our paper presents a number of problems and solutions that can be evaluated in polynomial time over the BGV ciphertext space. Additionally, we also survey other solutions that are better optimized for slight variations to the ciphertext space (such as those that utilize encrypted binary strings, as opposed to pure integers). The results of this survey conclude that it is still possible to compute an arbitrary circuit in polynomial time with the BGV scheme, however it is not necessarily efficient, nor is it feasible for some very costly solutions.

The Jin-Yi Cai lecture notes demonstrate that all polynomial time algorithms can still be computed in polynomial time using the fully homomorphic encryption scheme of Brakerski et al, despite many of them taking quite large polynomial amounts of time. Sorting a list of m elements takes time proportional to m^8, while a general polynomial algorithm with complexity $O(n^d)$ on a Turing machine (or any modern computer architecture) is not expected to be implementable with time less than $O(n^{5d})$ using the BGV encryption scheme.

Additionally, experiments carried out by Togan and Plesca indicate that comparisons over encrypted binary strings take time $O(n \log n)$ [14]. Results like these are not surprising, as it is necessary to optimize the multiplicative depth of operations over any fully homomorphic encryption scheme in order to make any arbitrary algorithm more efficient. In the case of universal computation in particular, it is necessary to design circuits of limited multiplicative depth to achieve feasible computation, otherwise incredibly large polynomials will surely dictate the upper-bound complexity of the algorithm.

5 Future work

We did not have time to test a physical implementation of our FHE operations utilizing the HELib library designed by Halevi and Shoup [10]. Though, the experiments of [14] are likely indicative of the complexities we should expect from our theoretical bounds. Asymptotic results can reveal a lot about the theoretical power of an algorithm, but they neglect important information such as the exact multiplier on the asymptotic time in the big O expression. For example, an $O(n^3)$ algorithm may be inefficient if it takes $1000000n^3$ steps to run and each step is a nanosecond. Additionally, the physical implementation may require optimizations or modifications to the theoretical algorithm that may increase (or decrease, with luck) the complexity of the expression. Our proposed algorithms all seem to be quadratic or cubic in complexity (when taking the bootstrapping case), but it is very possible that practical implementations of these proposed circuits may in fact be more inefficient than that, or are indeed quadratic or cubic for very large coefficients.

Additionally, it might be desirable to consider optimizing repeated or often used circuits for better multiplicative depth, or to support parallelization of multiplicative gates wherever possible, as this will reduce the complexity greatly (this can be seen in the binary tree method for multiplicative OR'ing that we have discussed earlier). Such optimizations should be the focus of development when considering the design of arithmetic circuits since multiplicative depth of the arithmetic circuits evaluated in a fully homomorphic scheme affects runtime so greatly. This can be seen in the works of [6], where simple implementations of past sorting algorithms have incredibly large multiplicative depth when compared to the algorithms proposed by Cetin et al that were designed to have reduced multiplicative depth. Such an approach to homomorhic algorithms is favorable if not necessary in order to produce feasible algorithms and protocols over the ciphertext space of any fully homomorphic scheme.

6 Appendix

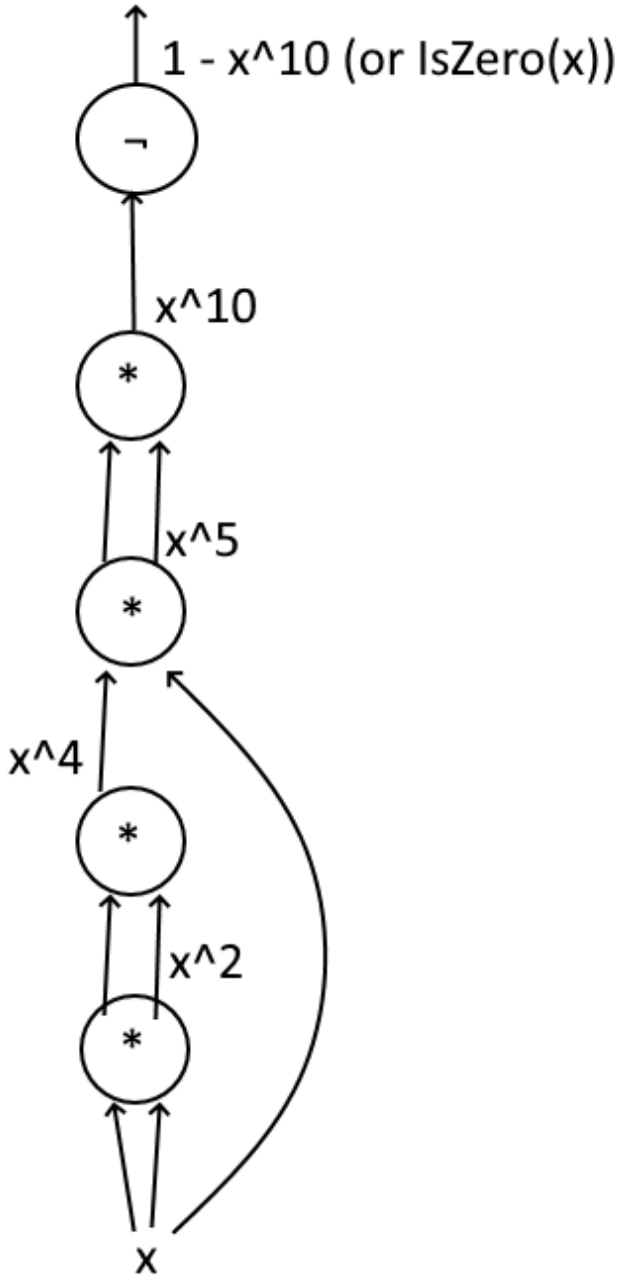

Figure 1: A diagram of the circuit for *IsZero* with $q = 11$, using the not gate from earlier

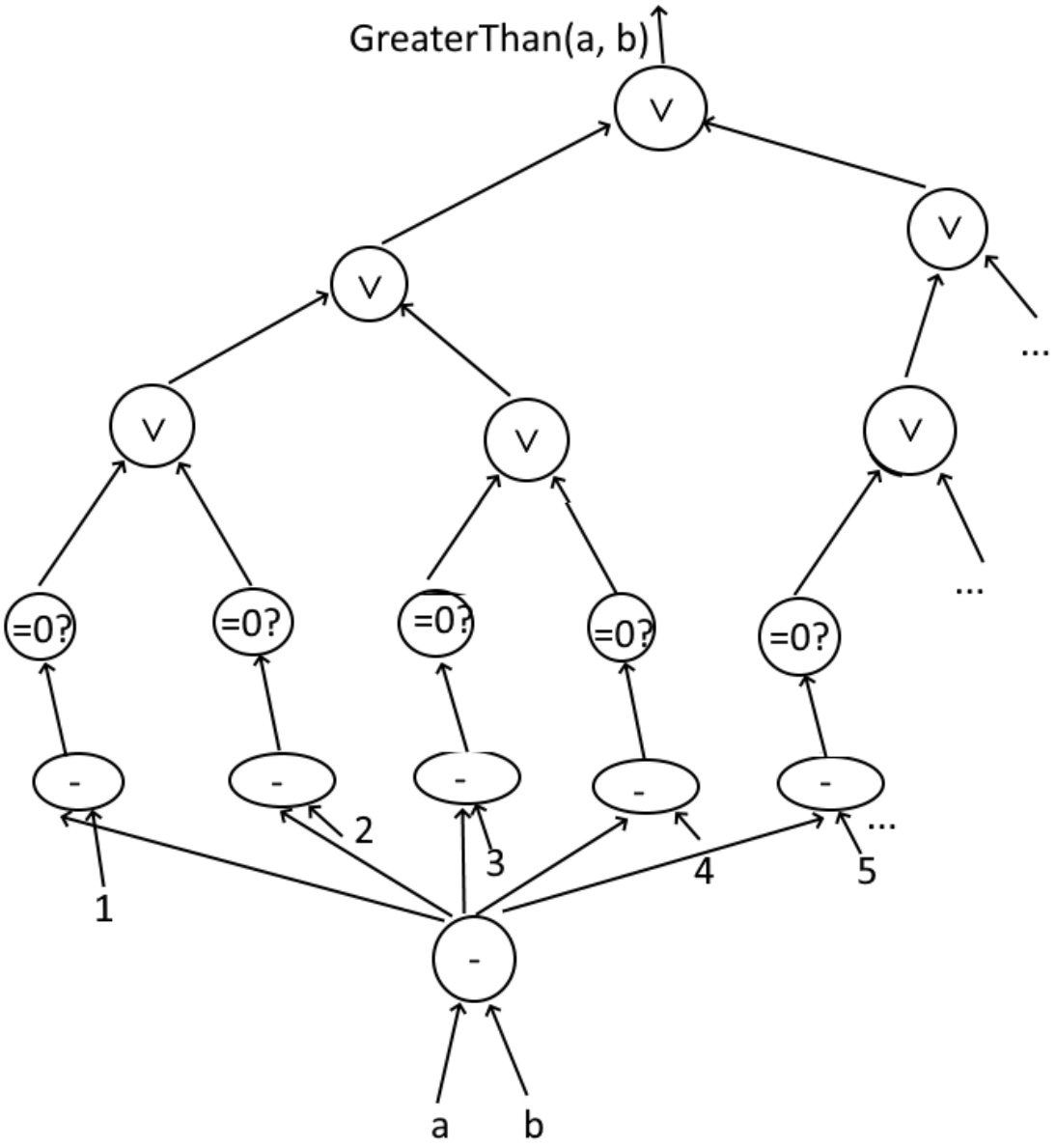

Figure 2: A diagram of the circuit for *GreaterThan*, using *IsZero* and subtraction gates

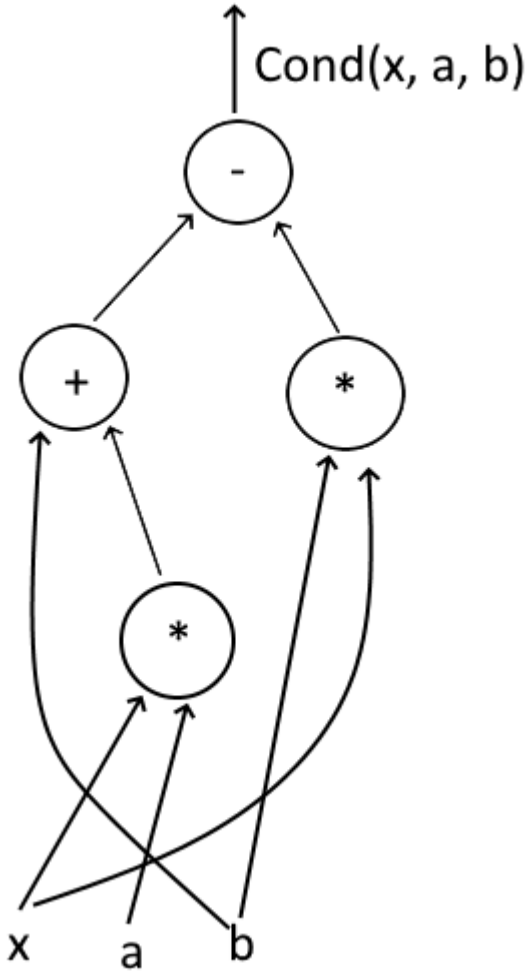

Figure 3: A diagram of the circuit for *Cond*

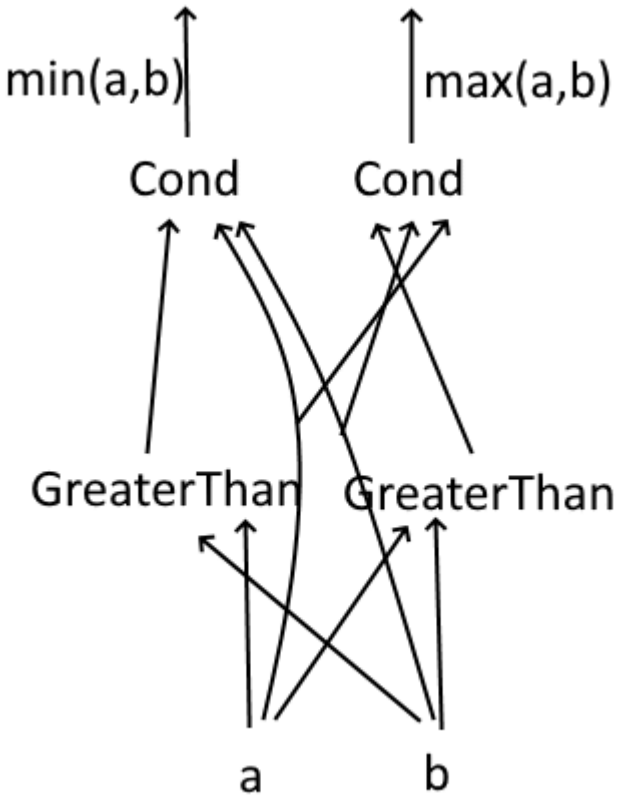

Figure 4: A diagram of the circuit for *Order*2

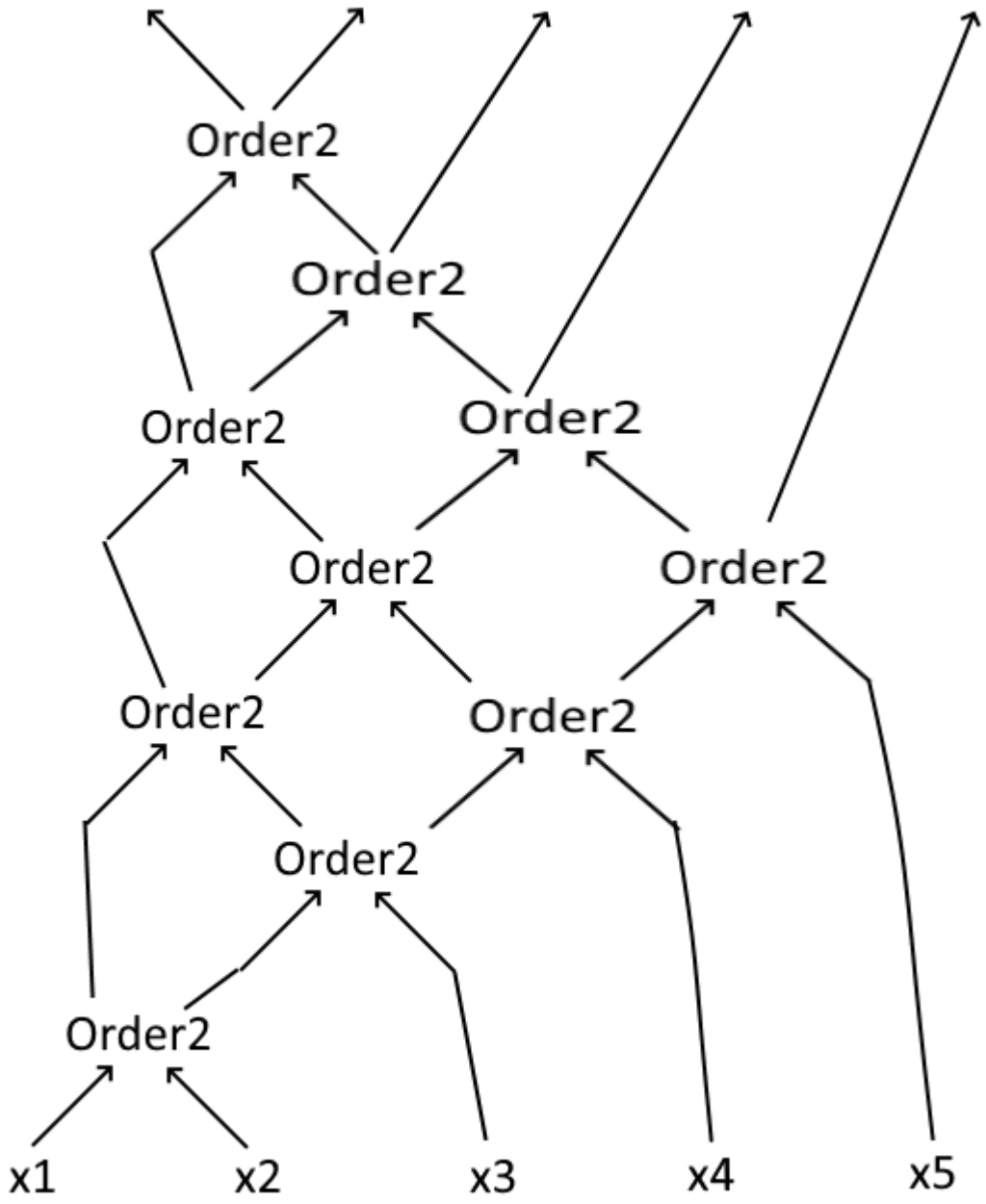

Figure 5: A full diagram of the circuit to sort 5 objects

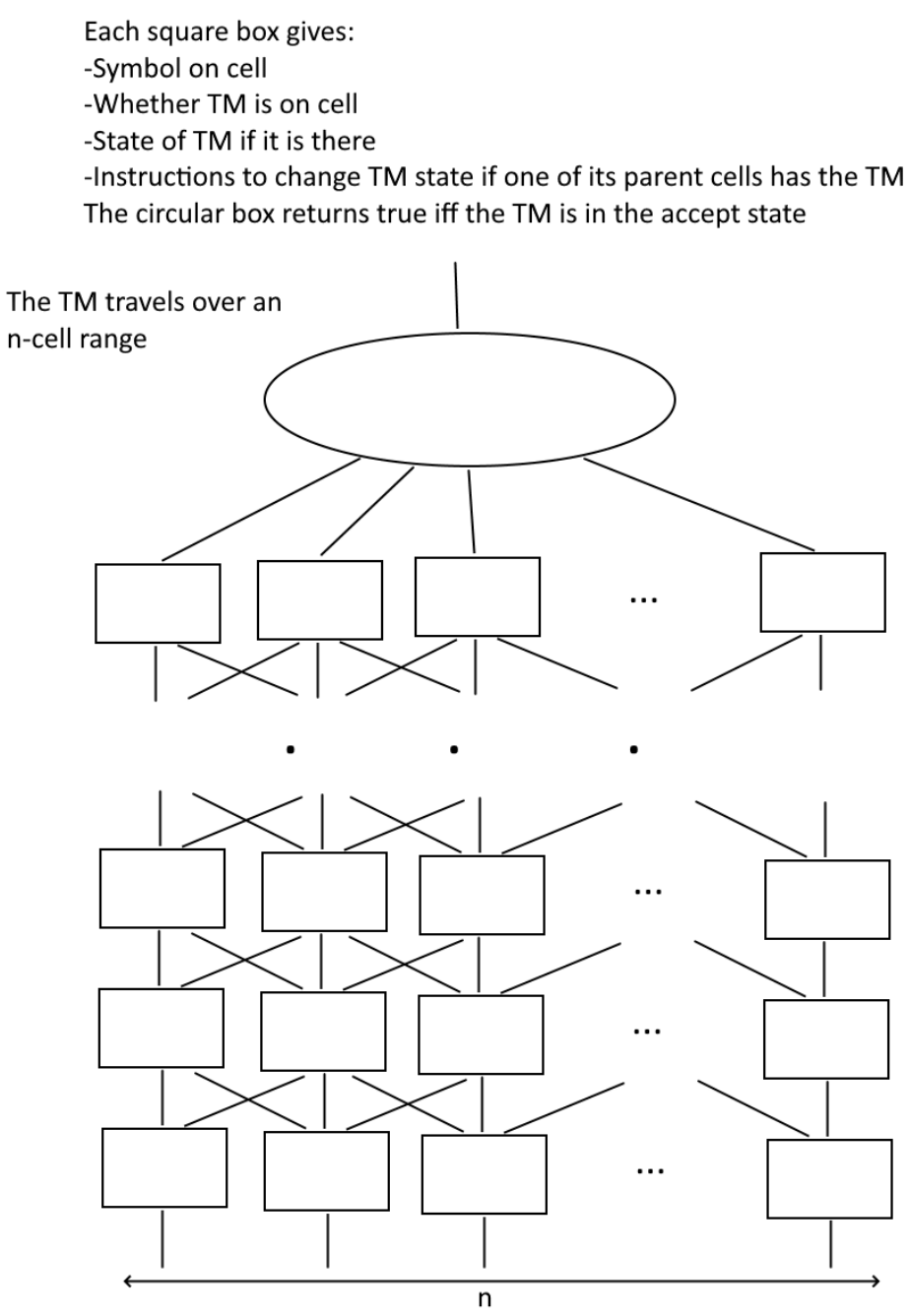

Figure 6: A description of how a circuit can simulate a Turing machine

References

[1] Frederik Armknecht et al. “A Guide to Fully Homomorphic Encryption”. In: *IACR Cryptol. ePrint Arch.* 2015 (2015), p. 1192.

[2] Daniel J. Bernstein. *Fast Multiplication And Its Applications*. 2003.

[3] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. “(Leveled) Fully Homomorphic Encryption without Bootstrapping”. In: *ACM Trans. Comput. Theory* 6.3 (July 2014). ISSN: 1942-3454. DOI: 10.1145/2633600. URL: https://doi.org/10.1145/2633600.

[4] Michael Brenner, Henning Perl, and Matthew Smith. “Practical Applications of Homomorphic Encryption”. In: *SECRYPT*. 2012.

[5] Jin-Yi Cai. “Lecture 09: Sparse sets and Polynomial-Size Circuits”. In: *CS 810: Introduction to Complexity Theory* (Feb. 2003).

[6] Gizem S. Cetin et al. “Low Depth Circuits for Efficient Homomorphic Sorting”. In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 274. URL: https://eprint.iacr.org/2015/274.

[7] Marten van Dijk et al. “Fully Homomorphic Encryption over the Integers”. In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–43. ISBN: 978-3-642-13190-5.

[8] Craig Gentry. “Fully Homomorphic Encryption Using Ideal Lattices”. In: *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. STOC ’09. Bethesda, MD, USA: Association for Computing Machinery, 2009, 169–178. ISBN: 9781605585062. DOI: 10.1145/1536414.1536440. URL: https://doi.org/10.1145/1536414.1536440.

[9] Shai Halevi. “Homomorphic Encryption”. In: *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*. Ed. by Yehuda Lindell. Cham: Springer International Publishing, 2017, pp. 219–276. ISBN: 978-3-319-57048-8. DOI: 10.1007/978-3-319-57048-8_5. URL: https://doi.org/10.1007/978-3-319-57048-8_5.

[10] Shai Halevi and Victor Shoup. *Design and implementation of HElib: a homomorphic encryption library*. Cryptology ePrint Archive, Report 2020/1481. https://ia.cr/2020/1481. 2020.

[11] Liam Morris. “Analysis of Partially and Fully Homomorphic Encryption”. In: 2013.

[12] Payal V Parmar et al. “Survey of various homomorphic encryption algorithms and schemes”. In: *International Journal of Computer Applications* 91.8 (2014).

[13] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. “On data banks and privacy homomorphisms”. In: *Foundations of secure computation* 4.11 (1978), pp. 169–180.

[14] Mihai Togan and Cezar Pleşca. “Comparison-based computations over fully homomorphic encrypted data”. In: *2014 10th International Conference on Communications (COMM)*. 2014, pp. 1–6. DOI: 10.1109/ICComm.2014.6866760.

[15] Mark A. Will and Ryan K.L. Ko. “Chapter 5 - A guide to homomorphic encryption”. In: *The Cloud Security Ecosystem*. Ed. by Ryan Ko and Kim-Kwang Raymond Choo. Boston: Syngress, 2015, pp. 101–127. ISBN: 978-0-12-801595-7. DOI: https://doi.org/10.1016/B978-0-12-801595-7.00005-7. URL: https://www.sciencedirect.com/science/article/pii/B9780128015957000057.